To Libby

Acknowledgements

There are a number of people who I would like to thank for their help during the time that it has taken me to write this thesis.

My supervisors, Professor Mark Cross and Doctor Steve Johnson for their invaluable support and guidance.

My colleagues, Chris Bailey, Peter Chow, Nick Croft, Emyr Evans, John Ewer, Yvonne Fryer, Cos Ierotheou, Peter Lawrence, Peter Leggett, Miltos Petridis and Chris Walshaw, for their help and patience in assisting me to write this thesis.

The staff and researchers at the School of Computing and Mathematical Science for providing a pleasant working environment.

The Engineering and Physical Science Research Council for supplying the funding that allowed me to escape from the pressures of industry and rediscover the world of academia.

Abstract

The motivation of this thesis was to develop strategies that would enable unstructured mesh based computational mechanics codes to exploit the computational advantages offered by distributed memory parallel processors. Strategies that successfully map structured mesh codes onto parallel machines have been developed over the previous decade and used to build a toolkit for automation of the parallelisation process. Extension of the capabilities of this toolkit to include unstructured mesh codes requires new strategies to be developed.

This thesis examines the method of parallelisation by geometric domain decomposition using the single program multi data programming paradigm with explicit message passing. This technique involves splitting (decomposing) the problem definition into P parts that may be distributed over P processors in a parallel machine. Each processor runs the same program and operates only on its part of the problem. Messages passed between the processors allow data exchange to maintain consistency with the original algorithm

The strategies developed to parallelise unstructured mesh codes should meet a number of requirements:

The algorithms are faithfully reproduced in parallel.

The code is largely unaltered in the parallel version.

The parallel efficiency is maximised.

The techniques should scale to highly parallel systems.

The parallelisation process should become automated.

Techniques and strategies that meet these requirements are developed and tested in this dissertation using a state of the art integrated computational fluid dynamics and solid mechanics code. The results presented demonstrate the importance of the problem partition in the definition of inter-processor communication and hence parallel performance.

The classical measure of partition quality based on the number of cut edges in the

mesh partition can be inadequate for real parallel machines. Consideration of the topology of the parallel machine in the mesh partition is demonstrated to be a more significant factor than the number of cut edges in the achieved parallel efficiency. It is shown to be advantageous to allow an increase in the volume of communication in order to achieve an efficient mapping dominated by localised communications. The limitation to parallel performance resulting from communication startup latency is clearly revealed together with strategies to minimise the effect.

The generic application of the techniques to other unstructured mesh codes is discussed in the context of automation of the parallelisation process. Automation of parallelisation based on the developed strategies is presented as possible through the use of run time inspector loops to accurately determine the dependencies that define the necessary inter-processor communication.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 The Nature of a Parallel Machine

The quest for greater performance has driven the development of computer technology at an exponential rate. Clock speeds and bus widths continue to increase while low power semiconductor technologies now permit Very Large Scale Integration (VLSI) to shrink the Central Processing Unit (CPU) of a 64bit computer onto a single silicon substrate. It has long been assumed that there is a fundamental limit to the performance that may be achieved by a single processor. How small can semiconductor features be made? How fast can a semiconductor switch operate? When does the technology reach a fundamental limit? [MF95]

Since the 1960's pipelined or vector processors have been at the heart of many supercomputers. Rather than operating upon a single variable at a time, these machines increase their computational performance by allowing a vector of data to be operated upon simultaneously [HJ81]. The achievable performance depends upon successfully loading the appropriate vector operands from memory [Rod82, Ier90]. Initially the vectorisation of code was an optimisation for the code author to implement. Subsequent development led to the vectorising compiler which could automatically extract the vector parallelism from the source code [DLD93].

An extrapolation of this concept led to the development of the array structured Sin-

gle Instruction, Multiple Data (SIMD) [Fly72] parallel machines in which whole fields of a variable could be subjected to the same operation in parallel [HB84]. These machines possessed large numbers of small processors (64 in Illiac-IV circa 1970, 65536 in DAP circa 1980) and gave rise to the description Massively Parallel Processing (MPP). SIMD machines have changed little since their conception and can still sustain a credible throughput in comparison with more modern architectures. Like the vector machines, they rely on running a code which maps well to the machine [Par82]. In this case a regularly structured code containing few inherently serial operations is required. Unlike the vector machines, automatic compilation of serial code for SIMD processing has not been possible. Mapping of irregular problems to efficiently utilise the power offered by SIMD machines has consequently been the focus of much research [Far89, FFL93, Wil91]. The difficulties encountered in successfully programming for SIMD has contributed to the architecture falling from popularity.

The notion that it may be more worthwhile to build a number of modest individual computers rather than one large one is not new. Many such parallel machines have now been successfully built, used and become obsolete [TW91]. Such machines are categorised as Multi Instruction, Multi Data (MIMD) [Fly72], of which there are two main variants: Distributed Memory (DM), in which each processor is equipped with its own private memory and Shared Memory (SM), where the memory is common to all processors [AG94]. Now that integration density can place what was until very recently considered a supercomputer onto a single chip, and furnish it with a quantity of memory in a similarly small space, with sufficiently low energy requirements to allow the intimate connection of many processing elements, this makes highly parallel MIMD the probable architecture for the next generations of supercomputers [FWM94].

The von Neumann programming model of a computer has not changed during these developments [vN66]. Programs continue to be written as a series of instructions to be executed in sequence. Indeed many algorithms depend upon the sequential order of variable evaluation. A diversity of new languages and paradigms have consequently been developed that attempt to express and exploit parallelism with concepts such as

Communicating Sequential Processes [Hoa86], tasks (Ada, Occam), data flow [DeC89] and data parallelism (FortranD, HPF) [vH92, Ric95]. There exists, however, not only a legacy of software that has been written in a simple sequential procedural manner, but also a large base of program developers who have no interest in parallel processing. Program developers are content with the von Neumann model as a means of algorithmic expression and want nothing more than a larger, faster serial processor. A means of efficiently mapping existing and future software onto DM MIMD platforms is therefore required. The success of the vectorising compilers has led to an expectation that parallelising compilers will eventually be produced [ZC90, CBB+94]. Success has been shown with automatic parallelism for shared memory parallel MIMD systems with small numbers of processors (Cray Y-MP, C90 (actually shared memory vector parallel), SGI Power Challenge, Sun Sparc20MP, Digital 8400) [Sun94]. But shared memory is unlikely to be feasible for large numbers of processors as the memory bandwidth does not scale with the number of processors. Virtual shared memory systems that allow distributed memory to appear as shared memory have shown some limited success (Kendall Square KSR1, Cray T3D) but fail to reach the potential peak machine performance largely as a consequence of the high degree of inter-processor communication required to sustain memory/cache coherence [Bom93]. The advantage of distributed memory is freedom from the SM bandwidth problem as the DM bandwidth scales automatically with the number of processors. This is seen to outweigh the disadvantage of having to explicitly express the distribution, communication and synchronisation of data between processors. The argument for DM MIMD is essentially an economic one. An enormous amount of development is directed towards the cost-effective high-performance workstation market. No matter how powerful these machines become there will always exist users who seek greater processing power. The simple interconnection of workstation technology allows the DM MIMD parallel machine to capitalise on the economy of scale of workstation development and provide the required power at a cost which is highly competitive in comparison with other High Performance Computing (HPC) technologies [Smi90]. The number of floating point operations (Flops) per dollar has become a new yardstick for

the performance measurement of HPC.

1.2 The Nature of an Unstructured Mesh Code

Computational Mechanics (CM) may be applied to the modelling of diverse physical systems (structural mechanics, structural dynamics, fluid dynamics, electromagnetics, magnetohydrodynamics, etc.). The technique of applying a system of equations over a discretised domain leads inevitably to the concept of a mesh or grid. A mesh describes the spatial nature of a discretisation. Wherever possible this thesis will deal with 3 dimensional space, this is however not always convenient for the purposes of illustration or example, where 2 dimensional space will normally be used for clarity.

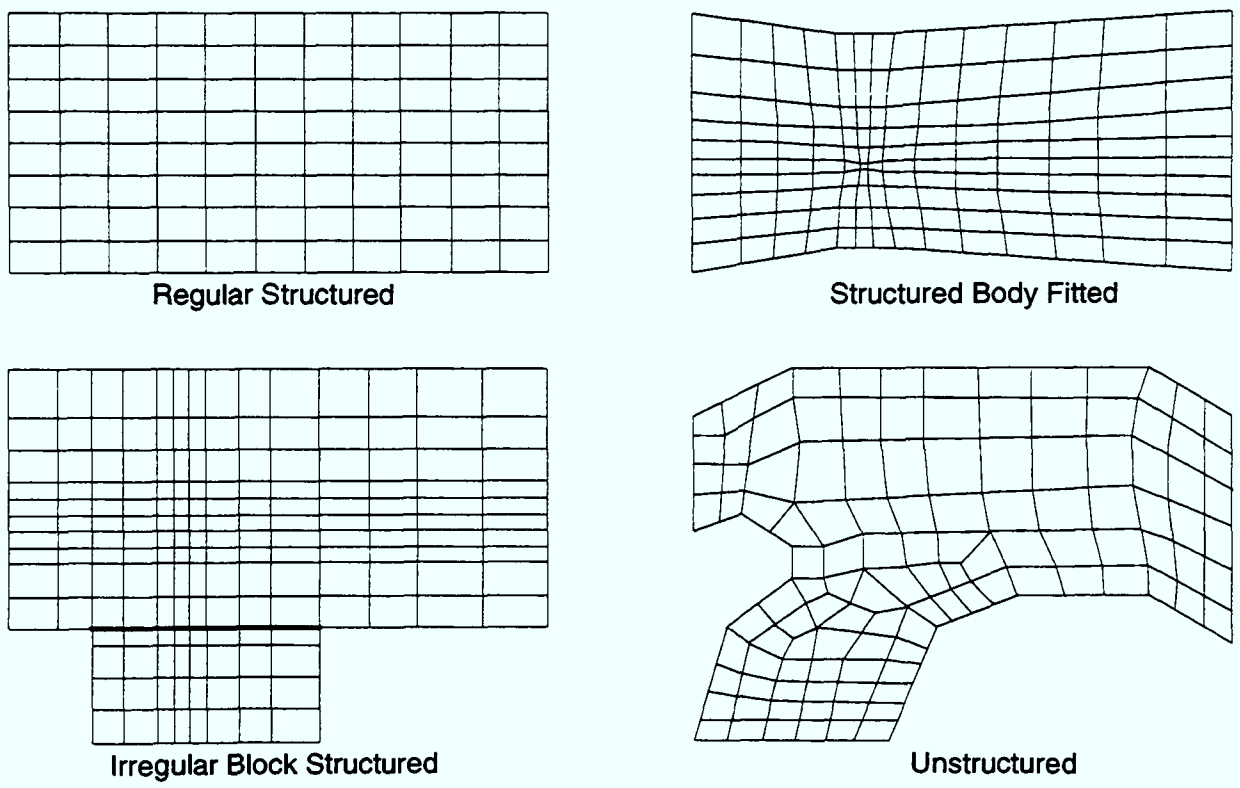

Figure 1.1: Four mesh categories.

The complexity of a computational mesh ranges from the simple regular structured to fully unstructured. Structured grids, suitable for transport phenomena modelling, were widely used in the development of Finite Volume (FV) (finite difference / control volume) schemes for Computational Fluid Dynamics (CFD) [Pat80]. Irregular and block

structured grids were introduced to allow FV schemes to work with complex geometries and a deformable mesh. The Finite Element (FE) method for structural and thermal analysis introduced an unstructured mesh to represent arbitrarily complex geometries [Zie77]. The desire to analyse flow in complex three dimensional geometries motivated the development of FE-CFD codes [MSSP88]. Difficulties with continuity and convergence in FE-CFD [Che91] led to recent work extending FV methods to unstructured grids [Cho93] and solid mechanics [FBCL91, CBCP92]. Unstructured mesh codes are unlikely to offer the computational efficiency of structured mesh codes. The implicit nature of a structured mesh avoids the need for indirection in variable addressing and allows great efficiency of coding, cache utilisation and vectorisation. But unstructured meshes provide a far greater flexibility for the modelling of complex geometries and avoid the need for the complexity of a block structured code. Now that automatic generation of complex unstructured meshes has become readily available [Law94] the focus of development is towards unstructured mesh codes.

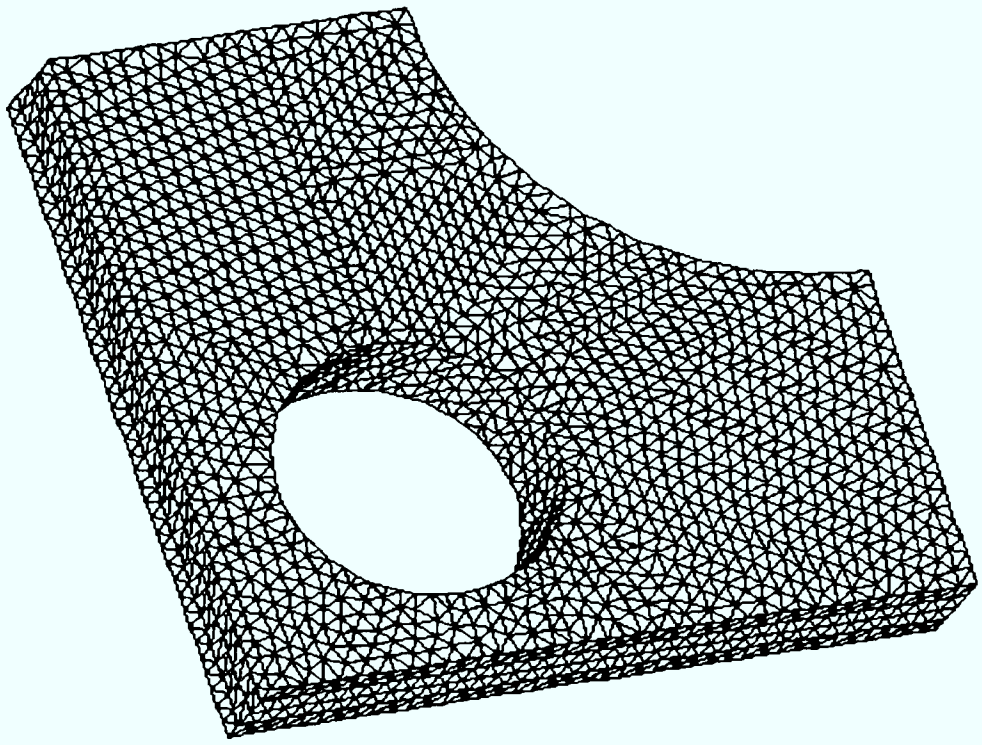

Figure 1.2: Automatically generated three dimensional unstructured mesh.

In parallelising a program the concern is not so much with the nature of the algo-

i) Minimise the changes to the original algorithm:

The parallel code should ideally produce identical results to the original serial code. This can be a necessary requirement for acceptance by code users who are familiar with the serial code and require confidence that the results generated by the parallel code execution are every bit as reliable as those produced by the serial code.

ii) Minimise the visibility of the parallel code:

The parallel code should be hidden from both the serial code developers and the parallel code users. This permits transparent maintenance of the parallel code alongside the serial code by the serial code developers. In addition this avoids deterring users from the parallel code. Code developers and users may be safely assumed to have no interest in parallelism and a significant interest in rapid code execution.

iii) Maximise parallel efficiency:

The parallel code must show significant speed-up over the serial code. The primary motivation for parallelisation is to reduce the code run-time. The parallel code must therefore use the parallel machine efficiently, otherwise the time and money expended on a parallel machine would be better invested on one or more serial machines.

iv) Portability to most DM MIMD platforms:

Parallel code needs to make good use of most currently available hardware, the DM MIMD model provides an efficient lowest common denominator hardware model. A programming model is therefore also required to necessitate only the most primitive platform support without loss of efficiency.

v) Scalability of computation:

DM MIMD Massively Parallel Processing (MPP) is the direction in which the high Flop per Dollar supercomputers are being developed. Although there continues to be much discussion concerning the implementational details of such MPP's, the

development of high performance, highly integrated serial processors will inevitably lead to the interconnection of increasing numbers of such processors (Cray T3D, Intel Paragon, IBM SP2, TMC CM5). To take advantage of the full power of MPP's the performance of a parallel code needs to be able to scale with the number of available processors. Doubling the number of processors should ideally halve the run-time.

vi) Scalability of memory:

Larger machines allow larger problems to be solved. To make full use of the distributed memory a parallel code must be able to distribute a problem over the DM machine. Globally dimensioned data items (data objects that are not distributed) must therefore be avoided.

vii) Automate the parallelisation process:

The human effort required to parallelise a CM code is significant. The majority of this effort is demonstrably automatable for structured mesh codes [JICL94, CIJL94]. A strategy is required which can minimise human intervention in the process of parallelising unstructured mesh based codes.

1.4 Parallelisation Strategies

Why use a parallel processor? Why not simply use many serial processors? There are two significant reasons; one is to provide a machine which can sustain a problem size that is too large to fit onto a serial processor, an other is to reduce the critical path to a solution. Given a set of interrelated tasks, a task interaction graph can be produced to describe the operations required to find the solution. Tasks may be carried out in sequence, one after the other, or some tasks may be executed in parallel as concurrent processes. The greater the level of concurrency that can be employed the less time is required to achieve the solution. Parallelism in computation exists in many forms and many different approaches have been used to exploit the parallelism that can be found in CM codes.

Task farming, for example, has the advantage of potentially high parallel efficiency by keeping all processors busy. As soon as a processor completes one task another is initiated. The technique is however, only suited to problems which present a large number of unrelated tasks such as Monte Carlo techniques. To achieve any efficiency the amount of data to be sent to and returned from each task must be insignificant in comparison to the task computation, which for a CM code is unlikely.

Algorithmic parallelisation involves each processor operating on different parts of a algorithm. For example solving flow in three dimensions could be achieved by solving for each dimension on differing processors. Taking the example further other computed variables could be distributed over a set of processors. Each processor calculates its variable and hands the problem to the processor computing the next stage in the algorithm. This scheme has little to commend it as it suffers from a high communication requirement and poor efficiency as each stage in the calculation will take a different amount of time leaving most of the processors waiting for data.

Geometric decomposition partitions the problem space over a set of processors. Each processor executes the same algorithm on their own section of the problem. This method has the advantage of flexibility to allow variations on the decomposition strategy to be used to minimise the communication and maximise processor utilisation. Partitioning may be based on the mesh geometry or topology, or on the distribution of computational effort within the algorithms used in the code. For example computational partitioning of a CM code based around a direct solver may be dominated by the solver which dictates the decomposition of the problem. Often a wraparound partition of a matrix (i.e. with P processors, processor q owns matrix rows $q, P+q, 2P+q \cdots$) may be required to keep the processors busy in the solver. This can also determine how other parts of the mesh are to be distributed. For example in the FAMCALC parallelisation [JAC92] the finite elements are distributed in a wraparound fashion according to their inclusion in the system matrix. In this case a large communication overhead is incurred to allow satisfactory processor utilisation. As is often the case with CM codes based on short range interactions communication can be minimised and processor utilisation maximised

by a domain decomposition based on the geometry (topology) of the mesh.

1.5 Parallelisation by Domain Decomposition

Domain Decomposition (DD) is a generic name given to a variety of computational activities which involve the division of a problem space into two or more parts that may be operated on separately to some advantage. Such is the interest in DD that there is an annual conference devoted to domain decomposition methods in all their diversity [KX93]. Originally developed as a means of solving engineering problems that were too large to fit into machine memory [Kro63], there has been a revival of interest in domain decomposition as a means of mapping CM codes onto parallel computers [Wil90, BCG93]. Parallelisation by DD is a divide and conquer strategy in which a problem domain is decomposed into a set of sub-domains which can then be operated on in parallel. Attempts have been made at new parallel algorithms which seek to find a partial solution for each sub-domain and then reconcile the partial solutions across the sub-domain interfaces [FXR92, Lai95]. This runs contrary to the strategies discussed in this thesis which should meet objective (i) (and (ii)) and maintain as far as possible the integrity of the original algorithm across the partitioned domain. This thesis is concerned only with geometric DD as a method for the direct parallelisation of unstructured mesh based CM codes for DM MIMD computers. This is a technique that is well suited to the short range dependence typical of a CM iterative method (Section 1.2).

The initial step in applying DD to an unstructured mesh based code is to obtain a partition of the mesh that allows the problem to be distributed amongst the available processors in such a way as to equally apportion the computation time on each of P processors. If this process is 100% efficient then the processing time for a problem may be divided by P. To achieve a high parallel efficiency with a large P has consequently become the subject of much research. Much success has been shown with the parallelisation of structured grid codes using DD with message passing [JC91, GCC+93] , wherein the partition of the mesh is closely mapped onto the processor interconnection

topology in order to minimise the inter-processor communication. Some work on unstructured mesh codes following the same topology mapping principle has shown success [RL90]. A generic method that can provide good performance without requiring an absolute adherence to the processor topology is needed to allow automated decomposition of unstructured meshes with scalability and efficient portability.

A number of languages and environments have been developed for the generation of code which may be automatically parallel. Parallel languages have much to offer, but are of limited use for 'dusty deck' codes and more importantly of little interest to serial code developers. It is simply not acceptable to require code authors to learn new skills in order to be able to use parallel machines. It is a hard enough task to author a CM code in the first instance without having to spend more time and effort in persuading the code to run on a parallel machine. Environments and libraries for parallelisation may point the way for development of parallel code that is transparent to both the code developers and the code users, but they fall a long way short of addressing the entire parallelisation problem. The Computer Aided Parallelisation Tools project (CAPTools) at the University of Greenwich [JICL94, CIJL94] seeks to resolve the parallelisation of structured mesh Fortran codes through the use of an *interactive* toolkit based on highly sophisticated interprocedural dependence analysis. It is hoped that the strategies developed in this thesis will extend scope of the CAPTools package towards the parallelisation of unstructured mesh codes.

Chapter 2

Parallel Processing

A Distributed Memory Multi-Instruction Multi-Data (DM-MIMD) parallel computer is, in the simplest of terms, a number of interconnected processors, each of which is equipped with a quantity of memory. The combination of processor and memory is referred to as a Processor Element (PE). Programs (processes) running on the processors can communicate with each other in what has been described and formalised as concurrent communicating sequential processes [Hoa86]. In this way the processors operate in unison to provide a high overall rate of computation.

Many different approaches to programming for a DM-MIMD parallel machine have been explored [Kri89, LC90]. The parallel programming strategy used in this thesis is a Single Program Multi Data (SPMD) message passing paradigm. Each processor runs the same program (process) on its part of the data set communicating with other processors through the exchange of messages. The terms processor and process for the purposes of this thesis are consequently interchangeable. This strategy has similarities with the data parallel strategy [Hil94] but uses an explicit derivation the data partition based on the mesh. The strategy is actually a master slave scheme during input/output processes in that one processor is the designated master simply because it has control of the i/o processes. Parallel i/o hardware is still uncommon and any dependency on such platform specific features would pose a significant barrier to portability.

Any time spent in communication between the processors is an overhead not incurred

with serial processing and so to use a parallel machine efficiently the inter-processor communication must be minimised. Successful inter-processor communication requires a high degree of synchronisation between the processes [Val90]. Successful parallel processing requires that no processor needs to idle whilst waiting to synchronise with other processors. To achieve an efficient parallel implementation the workload must therefore be balanced amongst the processors.

2.1 Processor Interconnection

There are many varied and novel methods by which processing elements may be interconnected. The relative merits of the differing interconnection strategies are discussed at length by several authors [TW91, AG94, FWM94]. A number of interconnection topologies have been tried. The richly connected hypercube (nCUBE 2s), two and three dimensional arrays, often looped into a ring or torus connection (Intel Paragon, Cray T3D) and other connections such as fat trees (Thinking Machines CM5) have also been used [vanderSteen94]. The advent of the INMOS transputer [Inm89c, Inm89a] with four high speed serial communication ports integrated into a single chip CPU popularised the scheme of a simple interconnected mesh of relatively low cost, highly integrated PE's [HJ88]. The companion chip to the transputer family, the Inmos C004 32-way crossbar switch [Inm89c, Inm89b] provides at low cost a means of reconfiguring the interconnection topology of an array of transputers. This model has persisted into many new designs, most probably as a result of the low cost of implementation coupled with a potentially high performance. Different switching technologies have been employed (IBM SP2, NEC Cenju-3, Meiko Computing Surface) but the reconfigurable interconnection model remains largely similar. Consequently this is the model of PE interconnection that this thesis will focus upon. Because this model of a parallel machine relys upon no special features the concepts discussed will be applicable to the majority of DM-MIMD platforms. Highly sophisticated and complex processor interconnections suffer significantly from the high cost of implementation. To remain cost effective the interconnection cost

must be small in comparison with the PE cost. Additionally the reliance upon machine specific features in programming may provide a good performance on one platform but can result in restricted portability. Advanced interconnection features may be implemented on simple platforms through the use of a software communication harness, but with consequent performance degradation. To achieve a cost effective parallel machine the investment in processor interconnection must result in a well balanced ratio between the communication performance and the calculation performance of the individual PE's.

2.2 Inter-Processor Communication

The key parameters for communication between processors are the bandwidth of the communication channels and the startup latency time to send a message.

The bandwidth r_n is the rate at which a data packet of length n may be transferred between two processors, normally measured in millions of bytes (Megabytes) per second (MBs^{-1}). Typical bandwidths may be $1.7MBs^{-1}$ per connection for the T800 transputer up to $170MBs^{-1}$ per connection in the Intel Paragon. For clusters of workstations connected by ethernet TCP/IP the bandwidth is more like $0.9MBs^{-1}$ [DD95]. This bandwidth cannot however be shared simultaneously by all of the processors as they all share the same ethernet connection. A more meaningful measure of interconnect bandwidth may be to divide the sum of the bandwidth of all interconnects in the machine by the number of PE's to give the bandwidth per processor. Clearly the bandwidths provided by different parallel systems ranges dramatically over two orders of magnitude. This spread in performance is even wider if the bandwidth per processor is considered.

The definition of latency varies but should give some measure of the time that it takes for a communication or message to begin transmission [CDJ95]. Latency is usually measured in microseconds (μs)and varys markedly from around 3μs in the Cray T3D up to 900μs for ATM-100 TCP/IP [DD95].

Measurement of the peak achievable communication performance for a platform can be misleading. The nature of a parallel code is that execution is synchronised in data

exchanges [Val90]. Ergo the critical communication is not with one individual message in the machine but with every processor involved in communication. The effect of this on the actual communication performance is highly dependent upon the machine hardware implementation. None of the DM machines offer a totally interconnected processor network and hence the interconnection bandwidth is shared amongst the processors. A more meaningful measure of latency and bandwidth can be obtained with the processor interconnects saturated as this reflects more accurately the communication of a typical code execution [MWC+95]. It is possible to saturate the interconnects with either local (near neighbour) or distant (non adjacent) traffic which will give differing measures of communication performance. The degree to which this will affect measurement is of course system dependent.

The number of processors (hops) between the source of a message and its destination affects the time for a message to complete. Jack Dongarra [DD95] considers the per hop delay to be a linear function of distance and so gives a model of the time t_n required to transmit n bytes of data as:

$$t_n = \alpha + \beta n + (h-1)\gamma \tag{2.1}$$

With start up time (latency) α, per byte time β, per hop delay γ and number of hops h. The bandwidth of the system can therefore be expressed as:

$$r_n = \frac{n}{\alpha + \beta n + (h-1)\gamma} \tag{2.2}$$

Hence the peak bandwidth r_∞ of a system is therefore expressable as:

$$r_\infty = \frac{1}{\beta} \tag{2.3}$$

A popular measure of the communication performance that combines latency with bandwidth is the bisection bandwidth $n_{\frac{1}{2}}$ defined as the message length at which half of the peak bandwidth is reached (perhaps better described as the bisection message length). For a single hop message this reduces to being simply the ratio of latency to peak bandwidth:

$$n_{\frac{1}{2}} = \frac{\alpha}{\beta} \tag{2.4}$$

It can be useful to consider whether bandwidth or latency is the bound on the performance of a code on a particular platform. The latency is often large in comparison with the time to transmit an individual data item. Given that the most obvious optimisation is to communicate only the data that is absolutely necessary, the next step is to minimise the number of transmissions that need to be made. Bundling the data to be communicated into large packets that require infrequent transmission reduces the latency overhead but incurs the overhead of copying data into buffer space. The extent to which communication may be buffered depends upon the individual code.

A parallel machine may be characterised by the communication to calculation ratio. This is sometimes given as the ratio of the time to send a one word message to the time for a floating point operation [FJL+88]. The notion being that a machine is well balanced if this ratio is less than unity. The actual MFlop performance is seldom maximal. As processor clock speeds increase to rates well beyond the access times for Dynamic Random Access Memory (DRAM) cache success rate begins to dominate the returned processing speed. Communication performance is both code and problem dependent as to whether latency or bandwidth form the limit. The computation to communication ratio is consequently somewhat arbitrary and subjective but if considered carefully can give a reasonably meaningful comparison of machine performance [AG94, FWM94]. A high ratio is likely to give poor parallel performance, the inter processor communication causing a processing bottleneck. A very low ratio would suggest that the investment in communication outweighs the investment in processing. Isolated consideration of the achievable parallel efficiency or speed-up of an application may give a misleading impression of the machine performance. The users (purchasers) viewpoint is usually more pragmatic involving wall-clock and dollars [FJL+88].

2.3 Communication Model

2.3.1 Shared Memory

From a programming viewpoint the simplest communication model is the shared memory model in which the entire machine memory is considered to be shared by all processors. For a DM-MIMD machine this leads to a locality dependent Non-Uniform Memory Access (NUMA) which can be handled to a some extent by advanced compiler techniques [LP92]. Whilst this presents an attractive model for programming and is amenable to automatic parallelisation it is an inefficient model for communication, giving rise to many small communications and hence tending to be latency bound. Nevertheless this can be a moderately successful communication model for small to medium scale parallelism (2-16 processors) and low latency platforms.

2.3.2 Message Passing

Message passing provides an explicit control of the inter-processor communication in which data to be transmitted is considered to be a messsage sent to a destination processor. This allows greater optimisation of the inter-processor communication and consequently is the communication model adopted in this thesis.

A communication harness of some description is normally used to implement message passing. At its most primitive the harness allows message passing between directly connected processors. More usually some form of 'wormhole' routing is provided that allows messages to be sent from any processor to any other processor hiding the underlying processor interconnection from the programmer [NM93]. A per-hop cost penalty on non local message passing as discussed in Section 2.1 means that messages should be wherever possible nearest neighbour (localised) to maximise efficiency. Implementational details of the message passing paradigm vary greatly but may be contrived to provide a uniform view of the parallel machine across a wide range of platforms (Section 2.4.2). It is now widely accepted that shared memory offers a simple port to serial codes to attract code developers and users to parallel processing but cost effective efficiency can only be

obtained from low latency, high bandwidth, localised message passing.

2.4 Code Structure

Implementation of a message passing parallelisation into an unstructured mesh code must be largely hidden in order to comply with objective (ii). A structured approach to the parallel implementation can go a long way towards achieving this aim. The SPMD paradigm is used in this thesis as it allows a single source code parallel program to be developed which may be maintained as a serial code by the original code authors. The DD method adopted requires extension of existing data structures and additional data structures to define the mesh decomposition and inter-processor communication. These additional data structures need to circumvent the subroutine parameter lists to remain hidden. Include files containing common data areas provide a reasonably convenient way to manage these variables. Mapping of the partitioned mesh to the original mesh (required to rebuild partioned data for output) requires a global sized data structure that has to be distributed among the processors in order to remain scalable (objective (v)).

In this parallelisation strategy a shell structure illustrated in Figure 2.1 has been used to build layers of (in)visibility within the code. Around the outside of the shell are the majority of the original routines which remain unchanged.

At the next level in are the routines from the original code that have been modified to function in parallel. Most of these routines are changed only slightly in that additional subroutine calls have been included and some array dimensions and loop lengths are changed. The i/o routines unfortunately require extensive modification and remain a difficult area of code to successfully parallelise. Parallel i/o hardware is uncommon and so a serial pipelined approach has been adopted.

The visible parallel routines are provided by a parallel utilities library which provides routines that are locationless and directionless and so form a barrier to the visibility of the parallel implementation. At this level there is no concept of master or slave processor

or indeed processor number, position or communication channel. It is felt that the serial code developers should have no problem with this view of parallelism.

The communication library provides a barrier to the visibility of the parallel machine. The communication library consists a very simple set of communication routines used by the utility library to present a uniform functionality on all machines. This layer provides a portability interface and provides similar functionality to the many popular high level parallel communication harness' such as PVM or MPI.

The innermost level is the native communication harness provided for the parallel machine. Only the most primitive send and receive functions are necessary at this level thereby guaranteeing portability to most hardware platforms. Higher level communications at this level may however be used to simplify or improve the implementation of the communication library.

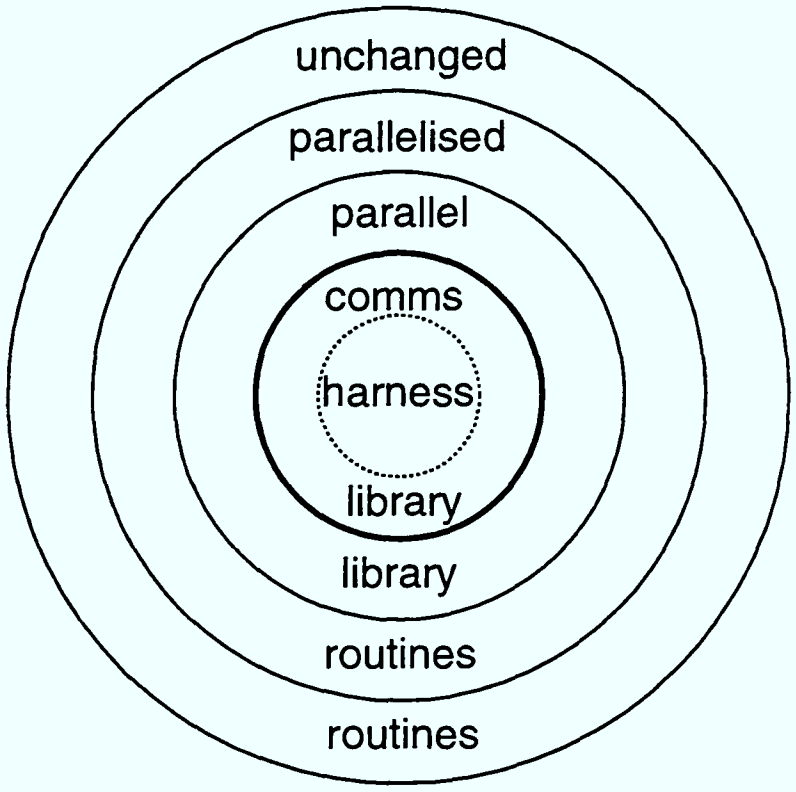

Figure 2.1: Shell structure of the parallel code.

2.4.1 Parallel Utility Library

Routines in the utility library are visible at the serial code level and must attempt to hide the parallel implementation whilst providing a parallel functionality which is conceptually straightforward. Simplicity of calling is of paramount importance in the library routines to achieve objective (ii). The routines in the library are described in Appendix A along with the parallel data declarations. The library is currently written in terms of the data structures used by the code being parallelised and hence is specific to that code. This library could however be made general purpose by adoption of a generic data structure for the utilities, this is discussed further in Chapter 6. The mesh decomposition routines at this level require extensive data structures and globally dimensioned variables. Embedding of these routines in the parallel code is not always possible, mainly due to memory restrictions. In which case they may be used to pre-process the serial problem files into a domain decomposed problem file that can then be used by the parallel program in place of the original problem specification. This process can be made reasonable seamless from the viewpoint of a code user.

Similar functionality has been developed for the Bulk Synchronous Parallel (BSP) [MR93] package and the Oplus package both from The Oxford Parallel group at the Oxford Computer Laboratory, LOCO from Katholieke Universiteit Leuven, PLUMP from CSCS in Switzerland [CDE+94] and DIME from Caltech [FWM94]. These packages offer a range of attractive features for portability, adaptive gridding and dynamic load balancing. The significant difference between their work and the work presented in this thesis is that they provide an environment and data structure that supports the *generation* of codes to handle irregular problems so that parallelisation of the code becomes more or less automatic. CM programmers cannot be expected to take on-board the overhead of authoring parallel code. This thesis therefore attempts a strategy for the parallelisation of *existing* codes for irregular problems with the intention of developing a methodology for automation of the parallelisation of old and new codes.

2.4.2 Parallel Communication Library

The parallel communication library imparts portability to the code by providing an interface between the parallel utility library and the machines' communication harness. Porting the parallel code to a new platform (harness) requires re-writing only the communication library. The library used for this thesis is the `CAPLib` library developed as part of the Computer Aided Parallelisation Tools project (CAPTools) at the University of Greenwich [CIJL94]. This library is constructed in two layers; `CAPLib` for high level routines and `CAPLow` for the low level portability shell. This further simplifies the portability of code using the CAP library system as only `CAPLow` requires porting. `CAPLib` is currently available for C Toolset on the Transtech Paramid, 3L Fortran on transputers, PVM2, PVM3 and MPI with Cray shared memory under development.

2.4.3 Communication Harness

A communication harness is in many ways analogous to an operating system in that it provides a means of loading an executable code onto the processors with a number of system facilities such as input/output. Most notably a parallel communication harness provides a means of inter-processor (inter-process) communication. Some manufacturers refer to their harness as a parallel operating system (Helios, Genesys, Parix) whilst others describe it more in terms of a loader or server program. In actuality it is usually a bit of both. Networks of workstations running UNIX can be configured as a Parallel Virtual Machine by using the popular PVM package or one of the more recently developed Message Passing Interface (MPI) packages. Some of the larger parallel machines use UNIX as the communication harness which then provides direct support for communication packages such as PVM or MPI but at the cost of a memory and processing overhead.

Communication Packages

The communication harness in Figure 2.1 may be implemented as any of a wide range of communication packages. There are almost as many different communication packages as there are parallel machines. An incomplete list of some of the most popular and

persistent of the packages is given here:

C Toolset - Inmos [Inm92]

PVM - Parallel Virtual Machine - Oak Ridge National Laboratory. [GBD+94]

MPI - Message Passing Interface - An international consortium coordinated through the University of Tennessee, Knoxville. [For94]

Parmacs - Parallel Macros for Fortran - Argonne/GMD. [Hem91]

CHIMP - Common High-level Interface for Message Passing - Edinburgh Parallel Computing Centre. [CTHW91]

PICL - Portable Instrumented Communication Library - Oak Ridge National Laboratory. [GHPW90]

Express - ParaSoft Corporation. [Par92]

MPL - Message Passing Library for the IBM SP2.

At the most fundamental level these packages provide a means of explicitly sending a message from one process (processor) to another. This simple message passing is all that is necessary for `CAPLib` to be ported to a communication package. Many of the packages provide more sophisticated features such as global commutative operations and asynchronous communications. Such features often rely on hardware specific calls for their successful implementation. Where available such features can be used directly by `CAPLib` to provide the functionality with consequent improved performance.

Communication Primitives

To achieve parallel message passing only a small number of communication primitives are required from the communication harness. Only Initialise, Send and Receive are actually required to implement a usable communication library. High level communication routines such as broadcast and global commutative operations can be built from these simple primitives. More efficient implementations of higher functions may be provided as primitives on some platforms and harness'. Some of the more sophisticated functions such as asynchronous communication must however be supported as primitives and cannot be built from synchronous communications. Primitive calls provided by the harness take many varied forms, some of the terms used to describe the routines are outlined below.

- synchronous (blocking) communication: returns when the operation is complete and data resources used in the call are available for re-use.
- asynchronous (non-blocking) communication: returns before the operation is complete and data resources used in the call are not available for re-use.
- broadcast: sends a data item to all processes
- reduction: performs a commutative arithmetic or logical operation on all processes.
- scatter: distribute a data item amongst the processes.
- gather: rebuild a data item using components from many processes.

Chapter 3

Domain Decomposition

Decomposition of a mesh based domain into a set of S sub-domains that may be allocated to a set of P processors involves finding a partition of the mesh so that the amount of compute time on each processor is very nearly equal. Two schemes are popularly used. One is to divide the problem into as many sub-domains as there are processors, i.e. $S = P$, so that each processor is allocated one sub-domain. The other scheme is to divide the problem into more sub-domains than there are processors, $S > P$, so that each processor operates on one or more sub-domains. This latter scheme has some advantages for targeting an inhomogeneous compute platform such as a network of workstations, in which the PE's are workstations which may have not only differing characteristics, but may also be subject to other workloads. Such a scheme can provide an effective coarse grained dynamic load balancing mechanism necessary for successful use of shared facility networks [MJ95]. Such networks tend to be reasonably small scale ($\sim P < 32$), in which case the overhead of dynamic sub-domain allocation may allow an effective speed-up. This thesis attempts to propose a scheme which will scale to a highly parallel ($\sim P > 64$) homogeneous DM MIMD processor array and so the former $S = P$ scheme is advocated. The simpler $S = P$ scheme carries a lower sub-domain allocation overhead and so may achieve a greater overall efficiency. Also there is an overhead incurred for each cut edge of the mesh which is minimised by keeping $S = P$. Edge is used here in a graphical sense meaning a relationship between mesh entities that is cut if the entities are in different

sub-domains. Dynamic load balancing schemes may still be implemented as fine grained migration of the mesh entities between the sub-domains.

Partitioning of a *structured* mesh is a reasonably straightforward procedure of cutting the mesh along the grid lines (2D) or planes (3D) [JC91]. Achieving a precise load balance in this instance requires that the mesh size along the partitioned axis is a multiple of the required number of partitions. Obtaining a balanced partition of an unstructured mesh is potentially a more complex problem and the focus of considerable research.

In order to solve for the nodes and elements around the edge of each sub-domain data is required from the neighbouring sub-domains according to the stencil of data dependency as discussed in Section 1.2. This data may be communicated as required from the processor on which the neighbouring domain is calculated, but this can lead to an unnecessarily large number of small communications. The strategy adopted in this thesis is to extended each sub-domain to overlap its adjacent sub-domains. This is discussed in more detail in Section 3.3. Each processor can then solve for the problem inside its sub-domain using the variables held in the overlap layer. Variables in the overlaps are updated from variables calculated on other processors to maintain a solution consistent with the original serial code.

3.1 Representation of an Unstructured Mesh

An unstructured mesh is specified as a hierarchy of components or mesh entities, each of which may be regarded as a data object or structure which can be used to provide a spatial, geometric or topological reference to the variables used in a computational mechanics code.

The definition of an unstructured mesh begins with a set of grid points or nodes, each of which is defined by set of spatial coordinates. The grid points describe the geometric shape and physical size of the mesh. Points are also convenient to provide a spatial reference for dimensionally independent variables such as temperature or pressure.

Points can be connected to form a set of edges, faces or both edges and faces. In

three dimensions edges can be connected to form a set of faces. Edges in 2D and faces in 3D may be used to provide a spatial reference for flux variables such as current density.

The space enclosed by a set of edges or faces describes an element. Elements have a volume and may be used as a spatial reference for volumetric entities such as mass or heat.

The perimeter or surface of a mesh defines a boundary which can be usefully associated with some boundary condition. Boundaries may also be defined internally to a mesh.

A defined volume or area within the mesh can be defined as a domain which is subject to certain conditions such as being of a material with specified physical characteristics.

The entity relationship diagram for a three dimensional unstructured mesh as shown in Figure 3.1 has only these few components and yet the web of relationships is highly interconnected. In two dimensions there is no definition of a face and so the relationships are a little more straightforward. Not all of the entities or the relationships are mandatory and the relationships may be explicit or implicit. The actual entities and relationships used varies from code to code.

The connectivity or topology of the mesh is explicitly expressed as relationships between like or differing mesh entities. For example the elements may be described in terms of their nodes as a list of node numbers for each element. From this information the element connectivity (adjacency) may be derived as a list of element numbers for each adjacent element. There is a trade off to be made between the memory used for the storage of these relationships against the ease of calculation required within the code. The nature of the integration employed by CM codes is nearest neighbour. Evaluation of an element based variable may for example require the variable values for all neighbouring elements and the coordinates of the points that comprise those elements (see Figure 1.3 d). This example would require the element to element connectivity to find the neighbouring elements and the element to node relationship to find the nodes of the adjacent elements.

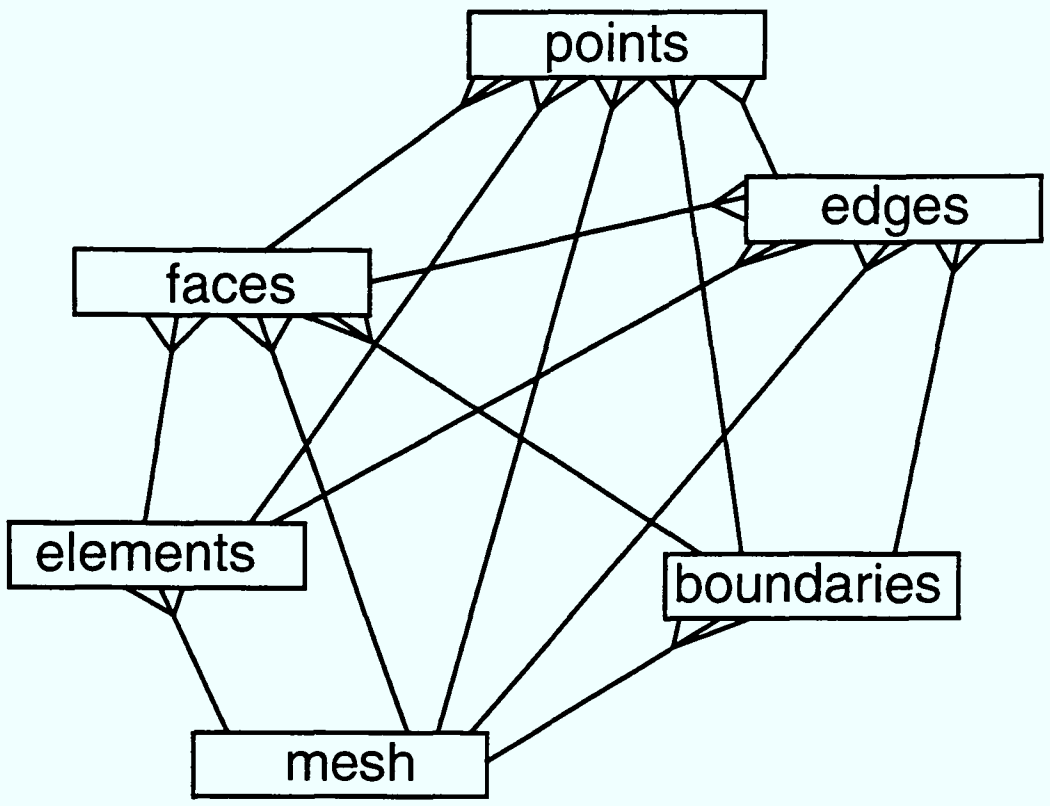

Figure 3.1: Entity relationship diagram for a three dimensional unstructured mesh.

3.2 Mesh Partitioning

The problem of partitioning an unstructured mesh has attracted the imaginations of many workers for more than twenty years [KL70] [PSL89] [BS93]. It is after all an interesting problem and one which at first sight at least seems well defined and self contained. A good mesh partition is one which divides the computational load equally amongst the sub-domains and minimises the amount of communication required between sub-domains. For many meshes it can be computationally prohibitive to find an optimal partition and computationally expensive to find a near optimal partition. On the other hand a reasonable partition may be calculated with little effort. The search for the 'best' partitioning algorithm has led to exploration of the middle ground, trading partition quality with the order of the partitioning routine.

Partitioning may be based on any of the mesh entities, usually either the elements or nodes of the mesh. A sensible choice is to partition according to the structure associated with the greatest amount of computation in the computational mechanics code. For

example a flow code dealing with element based variables would be partitioned according to elements whereas a stress code using node based variables would be partitioned as grid points. In actuality an element based code integrates over each face of each element and so a face based partition may be more appropriate. Similarly a node based code may integrate over each edge of the mesh and so an edge based partition may be more appropriate. The actual basis for partition chosen is not however of great consequence providing that the resulting mesh partition is balanced. This thesis will for simplicity normally refer to an element based partition. A mesh partition may be expressed in any of a number of ways, the method adopted is a simple list of the partition number for each element (entity). (Appendix B)

3.2.1 Load Balance

A fundamental objective in finding a partition is to balance the computational effort or load required in each sub-domain. The simplest approach is to assume that the load per element is homogeneous throughout the mesh. In this case the partition should have as near equal numbers of elements per partition as possible. Should the load be inhomogeneous then a weight or cost function may be applied to the elements to achieve a cost balanced partition. For example the computational effort required for each element may be proportional to the number of faces the element possesses. So tetrahedra will incur a cost of 4, bricks a cost of six and so on. An important consideration in load balancing is that it is not so much essential to achieve a totally uniform balance of load but rather that no one processor should have significantly more than average load. Any processor with an exceptional work load will cause all other processors to incur idle time with resultingly poor parallel performance. Should any one processor have too little work this will not hold up any other processors and have a correspondingly less detrimental effect on overall performance. This is illustrated in Figure 3.2 where the overall run time for partition A is longer than the overall run time for partition B despite the greater imbalance between the individual processor run times for partition B. The definition of a good load balance must reflect this effect. What is required is not a small deviation of

3.2.3 Processor Topology Mapping

The complexity and therefore the cost of building a totally interconnected non-blocking processor array is significant and so some form of interconnection map is generally favoured. As discussed in Section 2.1 this may be anything from a simple 1D or 2D array up to a 3D torus array or a fat tree structure. Many transputer based systems employ the Inmos C004 32 channel crossbar switch programmable link router chip allowing reconfigurable topologies to be constructed from a set of compute nodes. The IBM SP2 and the NEC Cenju3 use 4x4 switches to similar effect. A more detailed description of a number of popular and esoteric hardware architectures may be found in [vdS94, TW91]. In spite of what hardware manufacturers may claim there will always be a distance related communication cost. This cost becomes more significant as the number of processors increases. No matter how the processor interconnection is realised, a parallel processor platform will incur some form of topological communication cost. It is inevitable that it is more efficient to communicate with neighbouring processors than with distant processors. Robinson and Lonsdale [RL90] suggest that communication costs may be reduced by interconnecting the processors to reflect the mesh partition as illustrated in Figure 3.3. It may not however be possible or practical to reconfigure a processor array to suit a given partition. A more generic, flexible and scalable scheme is to consider the processor topology to be fixed as, for example, a 2D or 3D grid. This processor interconnection topology can then be reflected in the mesh partition. A transputer based platform, for example, would require the partition to limit the number of adjacent sub-domains to four (a 2D grid or 4 dimensional hypercube), as this is the number of communication links on each transputer. To this end weights can be applied to the partition to discourage the separation of neighbouring elements onto non-neighbouring processors [Jon94, Wal95]. In practice it can prove impossible to force a partition to adhere to a processor map, but the closer the partition reflects the processor map the greater the potential efficiency of the partition. A number of workers attempt to incorporate the underlying machine topology into the partitioning process in order to produce a partition that can provide improved parallel performance

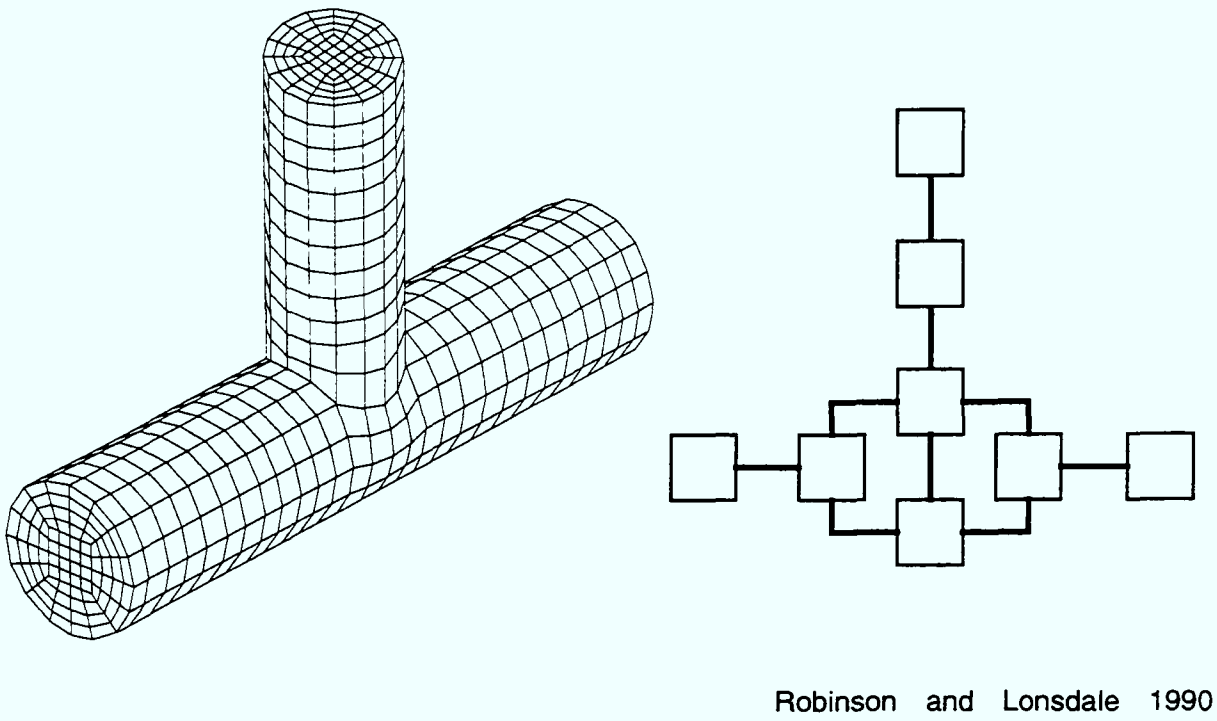

Figure 3.3: Processor interconnection mapped to a pipe mesh partition.

[Far89, WCE+95, Har94, MWC+95]. Figure 3.4 shows a mesh partitioned (using the JOSTLE code discussed in Section 3.2.4) into 16 sub-domains using three different partitioning strategies along with the corresponding processor interconnection graphs.

Regardless of how the mesh partition is calculated one is faced with the problem of mapping S partitions onto P processors ($S = P$) [SE87, SER90, BA92, HS92]. If P is small then all combinations may be tried to find the optimal mapping, that is the mapping which minimises the number of partition boundaries that do not align with processors interconnections. The combinations of mappings increase as P factorial which makes this impractical for even modest sizes of P. A simple scheme to obtain a mapping for little cost is to loop over all partitions in an initially arbitrary mapping looking for a partition which can be swapped so that communication cost reduction is maximised. This loop is iterated until no further cost reduction is found. Schemes such as this are prone to local minima traps but can give a useful mapping with little overhead [WCE+95].

3.2.4 Partitioning Algorithms

Some partitioning algorithms operate on the geometric mesh coordinates. Others treat the mesh as a graph $G(N, E)$ of nodes and edges. Graph based techniques have the advantages of dimensional independence and a true representation of the connectivity of the mesh in the partioning process. This is demonstrated by Nick Floros and Jeff Reeve to be of particular importance when partitioning highly complex shapes [FR94]. The graph to be partitioned may be simply the grid points (nodes) of the mesh or a dual graph of the mesh with the graph nodes representing for example elements and the graph edges representing the element adjacency. If the graph is based on elements of the same shape then the node degree (number of edges on each node) in the graph is more or less constant (nodes at the boundaries are of reduced degree). Partitioning to achieve an equal number of nodes in each sub-domain may achieve a good load balance. If however the graph is based on grid points, or the mesh is of mixed element shapes the node degree in the graph is variable. Partitioning a graph to achieve an equal number of edges (rather than nodes) in each partition may, in some cases, be more appropriate for load balance. Other factors may affect the computational load at each node of the graph, perhaps different materials, or phases for instance are associated with each node. Applying a weight to the nodes (perhaps based upon the number of connected elements and/or some other parameter) and then partitioning the weighted list can give an improved load balance. In practice it can prove difficult to accurately predict the computational load in each sub-domain.

Many of the schemes involve recursive bisections, variations on the bisection schemes involve cutting the mesh into more than two partitions at each step. This allows the algorithms to provide numbers of partitions other than 2^n.

What is required of a mesh partitioning algorithm is a high quality of partition at a low cost. The time required to calculate the partition must be insignificant in proportion to the time for the CM code to execute. High quality means a balanced load, short interfaces and a small number of interfaces. This paints a picture of partitions as uniform packed bubbles, shapes of minimum surface energy. Much of the current research

centres on hybrid approaches with graph reduction techniques and multilevel schemes to reduce the order of the problem [Jon94, WCE+95, HL93, VK95, DMM95, KK95]. A good but incomplete review of partitioning algorithms has been compiled by Chris Geenough [GF94] and Dirk Roose [RVD93]. A number of the algorithms have been collected into a package called RalPar [FG94]. Some of the more important techniques are covered in detail by Beryl Jones in her thesis [Jon94]. There follows a brief summary of many of the better known algorithms.

Recursive coordinate bisection

Recursive Coordinate Bisection (RCB) [Fox88] is a simple geometric scheme in which the grid points of the mesh are sorted into order along one axis (normally the longest) and then bisected. This process is repeated recursively on each partition until the required number of partitions is obtained. This gives rise to thin strip partitions with long interfaces. A variant of the scheme is Orthogonal Coordinate Bisection (OCB) in which the sort axis is alternated at each recursion. The resulting partitions are consequently more checkerboard in shape. An improvement is to bisect each partition along *its* longest axis, which is not necessarily the same for each partition.

Recursive inertial bisection

Recursive Inertial Bisection (RIB) is similar to RCB but bisects the geometric coordinates along the line of principal inertia [RVD93]. It can be expected that the line of principal inertia is aligned with the length of the mesh and the narrowest part of the mesh will be orthogonal to it. Whilst RIB is more expensive than RCB or OCB it is still a 'cheap' method and gives better results with concave geometries. RIB is still popularly used as it is fast and reliable.

Greedy

The greedy method is a graph based technique which begins with a node of minimum degree (minimum number of connected edges) and 'bites' level sets from the graph [Far88]

until the appropriate number of nodes ($\frac{N}{P}$) have been 'eaten'. This process is repeated on the remaining graph until all of the graph has been consumed. This is an extremely cheap method ($O(N)$) which produces mostly good partitions but is liable to leave some disconnected partitions (i.e. partitions that are split into two or more pieces).

MINCUT

MINCUT [KL70] employs heuristics to optimise a partition by swapping vertices of the graph between partitions to find the swap that minimises cost. "The general idea is to perturb the locally optimal solution in what we hope is an enlightened manner, so that an iteration of the process on the perturbed solution will yield a further reduction in the total cost." A logical exchange of all vertex pairs in the graph is performed and the effect of each exchange on the partition cost calculated. All exchanges up to the exchange that produces the minimum cost are then committed as actual exchanges. This process is repeated until no reduction in cost is obtained. This method attempts to climb out of a local mina trap but is not always successful.

Recursive graph bisection

Recursive Graph Bisection (RGB) [Sim91] is similar to RCB and RIB but operates on the graph of the mesh. A diameter of the graph is found and starting from one end of the diameter level sets are removed from the graph until the graph is bisected. The process is repeated recursively on each partition.

Recursive spectral bisection

Recursive Spectral Bisection (RSB) [PSL89] represents the graph with its Laplacian matrix $\mathbf{L}$. The method recursively partitions the graph by finding $\mathbf{x}$ which minimises $\mathbf{x}^T\mathbf{L}\mathbf{x}$. The eigenvector that corresponds to the second smallest eigenvalue (the first eigenvalue is trivial) is sorted and bisected to give a partition of the graph. This is a sophisticated and expensive method that provides a high quality partition that is especially suitable for complex geometries. Hendrickson and Leland [HL92] extended

the method to allow weighting of the nodes and edges and cutting into more than two partitions at each step. Multilevel Recursive Spectral Bisection (MRSB) dramatically speeds up the algorithm by coarsening the graph with clustering and using RSB on the coarsened graph [BS93]. This is a highly elaborate technique that provides the high partition quality of RSB at less cost.

Tabu search

Tabu search (TS) [Glo89, Glo90] is a combinatorial optimisation based iterative improvement technique that tries to avoid local minima traps by temporarily accepting unprofitable changes to the partition. Cycling in the search trajectory is avoided by keeping a history of the most recent changes, making further changes of the most recently moved nodes 'taboo'. Some open problems of TS are the determination of an appropriate 'prohibition period' and the robustness of the technique for a wide range of different problems. Some of the limitations of TS have been overcome in Reactive Tabu Search (RTS) [BT94] in which the appropriate size of the prohibition list is learned automatically by reacting to the occurrence of cycles.

Simulated annealing

Simulated Annealing (SA) is a generalised optimisation method that borrows ideas from a statistical mechanics approach to annealing in a cooling solid [KJV83, vLA87]. A parameter analogous to temperature is reduced during the course of the calculation. For each temperature a number of modifications to the current solution are tested. If a modification reduces the cost function the modification is accepted, otherwise the modification is accepted according to a probability function based on the exponent of the ratio of cost function to temperature. As the temperature cools the algorithm is less likely to accept a change that increases the cost. With a slow 'cooling' rate this method can produce good partitions but is computationally expensive. Developments of the basic ideas of SA have led to Mean Field Annealing (MFA) which combines SA type strategies with Neural Network techniques[BA92].

JOSTLE

JOSTLE [Wal95, WCE+95, MWC+95] is the code used to produce the partitions used in this thesis. The JOSTLE strategy is to derive an initial partition as quickly and cheaply as possible and then use optimisation techniques to improve the quality of the partition. Two alternative methods are provided to produce the initial partition. One method is a variation of the Greedy algorithm, in this case a graph based variant on the original mesh based algorithm proposed by Charbel Farhat [Far88]. The other method is geometric sorting which operates in a similar manner to OCB. This method provides a crude mapping to a $p \times q$ processor grid ($p \geq q$). The nodes are sorted on the longest axis and split into sets of N/p. The nodes in these sets are then sorted in the orthogonal axis and split into sets of N/pq. Having used one of the above methods to obtain an initial partition one of two optimisation methods can be applied to the improve the partition. Uniform optimisation is a technique in which each partition attempts to minimise its own surface energy analogous to the way that bubbles pack together. The technique works by calculating the centre of each partition in a graphical sense and determining the radial distance of each node from the centre. Nodes that are most distant from the centre can then be migrated between neighbouring partitions. Grid optimisation is a similar technique to uniform optimisation except that nodes are allowed to migrate only between neighbours in the processor grid. Four partitioning (mapping) strategies are provided by JOSTLE. *Unmapped* partitioning ignores the processor interconnection topology throughout the entire partitioning process. A *Postmapped* partition is an unmapped partition that has been mapped to the processor topology with a simple mapping algorithm applied post partitioning. The *Premapped* partition begins with a partition that is crudely mapped to the processor topology and then is optimised ignoring the processor topology to minimise the number of cut edges. The *Mapped* partition acknowledges the processor topology throughout the partitioning process. Some partitions produced by JOSTLE can be seen in Figure 3.4.

Strategy	Initial partition	Optimisation	Processor allocation
Unmapped	Greedy	Uniform	No
Postmapped	Greedy	Uniform	Yes
Premapped	Geometric sort	Uniform	No
Mapped	Geometric sort	Grid	No

Table 3.1: Partition mapping strategies provided by JOSTLE

3.2.5 Parallel Partitioning

Ideally the partition of the mesh should be carried out at run time in parallel. As P and N increase an $O(N)$ partitioning algorithm may become unacceptable for a solver running at $O(f(N)/P)$. Few of the available partitioning algorithms are suitable for parallel implementation. The work of Chris Walshaw [Wal95] and Ralf Diekmann [DMM95] aims to provide paralleliseable routines that can be used to partition and also re-partition meshes in a dynamic load balancing scheme. This strategy relies on obtaining a rapid initial mesh partition to crudely distribute the mesh across the processors and then operate on the partitions in parallel to optimise the partitions. Difficulties arise when the size of the mesh becomes too great to fit onto one processor. This is a natural consequence of massively parallel processing where the capacity of the whole machine may be orders of magnitude greater than the capacity of a single node. In such an instance the partitioning algorithm may have to begin by taking an arbitrary partition of the mesh as it is read in from file and distributed in sequence to a number (not necessarily all) of the processors. A high level of communication will then be required to re-distribute the mesh amongst all of the processors to provide a crude initial partition. If the partitioning strategy is, for example, to be the mapped JOSTLE scheme this will be a reasonably successful process. Geometric sorting will be a reasonably simple and cheap algorithm to implement as a parallel initial partition scheme.

iii) Determine the mesh overlaps to the neighbouring sub-domains.

iv) Re-number the mesh in each sub-domain.

v) Construct a template for overlap data exchange.

3.3.1 Derive Secondary Partitions

As mentioned in Section 3.2 the mesh entity that provides the dominant spatial reference used by the code to be parallelised is ordinarily chosen as a basis for mesh partitioning. This partition is referred to as the primary partition. Secondary partitions may be derived from the primary partition for the other mesh entities used in the code. The compute time for a CM code is dominated by the time spent in the solution of an equation of the form $\mathbf{Ax} = \mathbf{b}$. It is consequently important for load balance to obtain an equal number rows and an equal number of coefficients in each of the distributed $\mathbf{A}$ matrices. This inevitably results in some compromise. With an element based $\mathbf{x}$ for example, a primary partition based on elements will keep the vector length and hence number of rows in $\mathbf{A}$ balanced across each sub-domain. But the number of off diagonal coefficients in each $\mathbf{A}$ depends upon the number of internal faces in the sub-domain. Balancing elements will not necessarily balance matrix coefficients In the case of the two dimensional flow code used in this thesis the primary partition is based on elements and there is only one secondary partition, that being for grid points. For reasons of clarity the following discussion is based on an element based primary partition. The discussion is nonetheless applicable to other mesh entity partitioning orders.

Secondary partitions are inherited from the primary partition in accordance with the connectivity between the entities. For example, each node is connected to a number of elements, each of which belongs exclusively to one sub-domain. This provides a basis for the allocation of the node to a sub-domain. The most obvious and simple partition inheritance scheme is to allocate the node to the sub-domain which owns the majority of the connected elements. In the case of an equal number of connected elements being owned by two or more sub-domains, the node is allocated to the domain which

owns the least number of nodes. This simple, inexpensive scheme gives a good match between the primary and secondary partitions, but can lead to an unnecessarily high load imbalance in the secondary partition. It does not follow that two unstructured meshes with equal numbers of elements will have the same number of nodes, indeed there may be a large discrepancy between the two node counts. When the two meshes are sub-domains to be operated on in parallel this can produce an unacceptably high degree of load imbalance for element based matrix computations as discussed earlier and possibly even greater imbalance for node based calculations. If however the node allocation between the sub-domains is forced to be balanced the element and node partition may not be well matched which can result in an undesirably large and imbalanced overlap layer. This will consequently lead to large and unbalanced communications between the sub-domains. The comments about load and communication imbalance in sections 3.2.1 and 3.2.2 should be borne in mind at this point.

The load imbalance may be redressed to an extent by the use of more elaborate schemes to derive secondary partitions. A possibly superior partition inheritance scheme is to first locate the nodes for which all connected elements lie in one partition and for each node found, allocate the node to that partition. The remaining nodes are then allocated in turn to the least loaded domain beginning with the node which has the greatest connectivity to that domain.

It is conceivable that the nodal imbalance may become unmanageably large, in which case some nodes may require allocating to sub-domains that own none of the connected elements in order to redress the balance. This will result in a communication imbalance which may or may not be significant depending upon the characteristics of the hardware platform. The quality of the secondary partitions then becomes a platform dependent optimisation issue.

These schemes may be seen as an attempt at solving a graph problem by the application of simple heuristics. It may therefore be worthwhile to use graph based techniques to derive the secondary partitions. A possible scheme is to produce a weighted graph of the nodes which clusters the nodes for which all connected elements lie on one parti-

tion. This graph may then be partitioned using one of the graph partitioning algorithms developed for obtaining primary partitions. The work of Chris Walshaw [Wal95] is of interest here. The amount of effort that it is worthwhile devoting to the derivation of a secondary partition is problem dependent. Like the search for a primary partition there may be no singular optimal solution and a near optimal solution will in the majority of cases provide a sufficiently good solution.

3.3.2 Overlap Construction

The overlaps between the sub-domains are determined in accordance with the data dependency required by the code as discussed in section 1.2. For example, if the solution for an element based variable requires the values in all adjacent elements as illustrated in Figure 1.3a then the adjacent elements that lie in neighbouring sub-domains are added as overlaps to the list of elements. Similarly if the nodes that compose the overlap elements are also required as in Figure 1.3d then they too are added to the list of overlap nodes. In this way the description of the mesh for each sub-domain is extended to include all data that are required for solution of the sub-domain. The utility used to construct overlaps for the codes parallelised in this thesis uses a simple set of rules to determine the elements and nodes which are to be included in the overlaps (Appendix A).

When using only the element based flow and heat code;

Overlap elements are defined as:-

All elements that are adjacent to a core element.

Overlap nodes are defined as:-

Nodes of all elements including overlaps that are not core nodes.

However the node based stress code involves a more extensive data dependency and the required overlap layers become deeper so that;

Additional overlap elements are defined as:-

Elements that contain at least one core node.

Additional overlap nodes are defined as:-

Nodes that are connected to core nodes.

An example of the overlaps required for the flow code is shown in Figure 3.6. The same mesh is shown in Figure 3.7 with the additional elements and nodes in the overlaps required for the stress code .

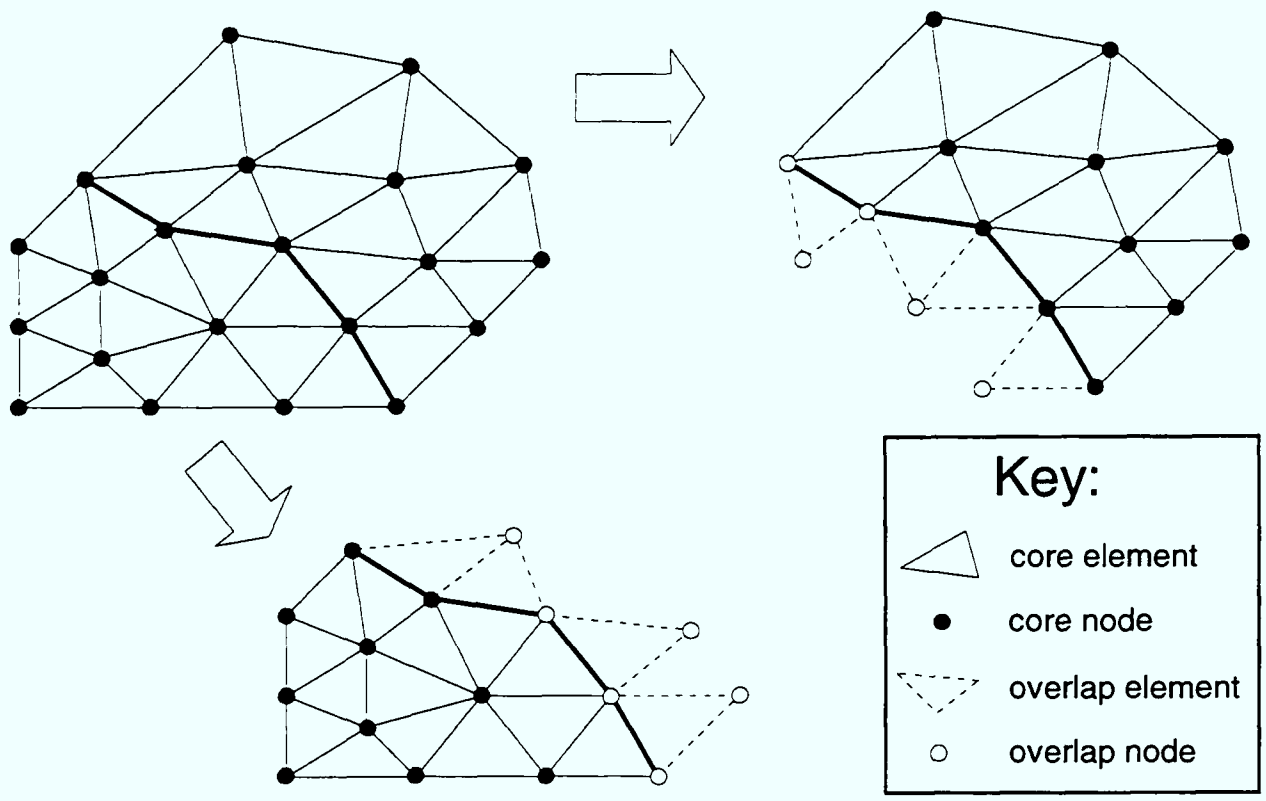

Figure 3.6: A mesh of 28 triangles divided into two sub-domains with the overlaps required for the flow scheme.

Providing that the mesh data structures are either one dimensional linked or indexed lists, or stored as multi dimensional arrays in which the number of entities is the highest index (last in F77, first in C) then the overlaps may be stored as extensions to existing data structures which allows them to be passed to subroutines and addressed in the parallel code in the same manner as the original data structures. This hides the parallelism and results in only small source file changes being required to extend mesh as it

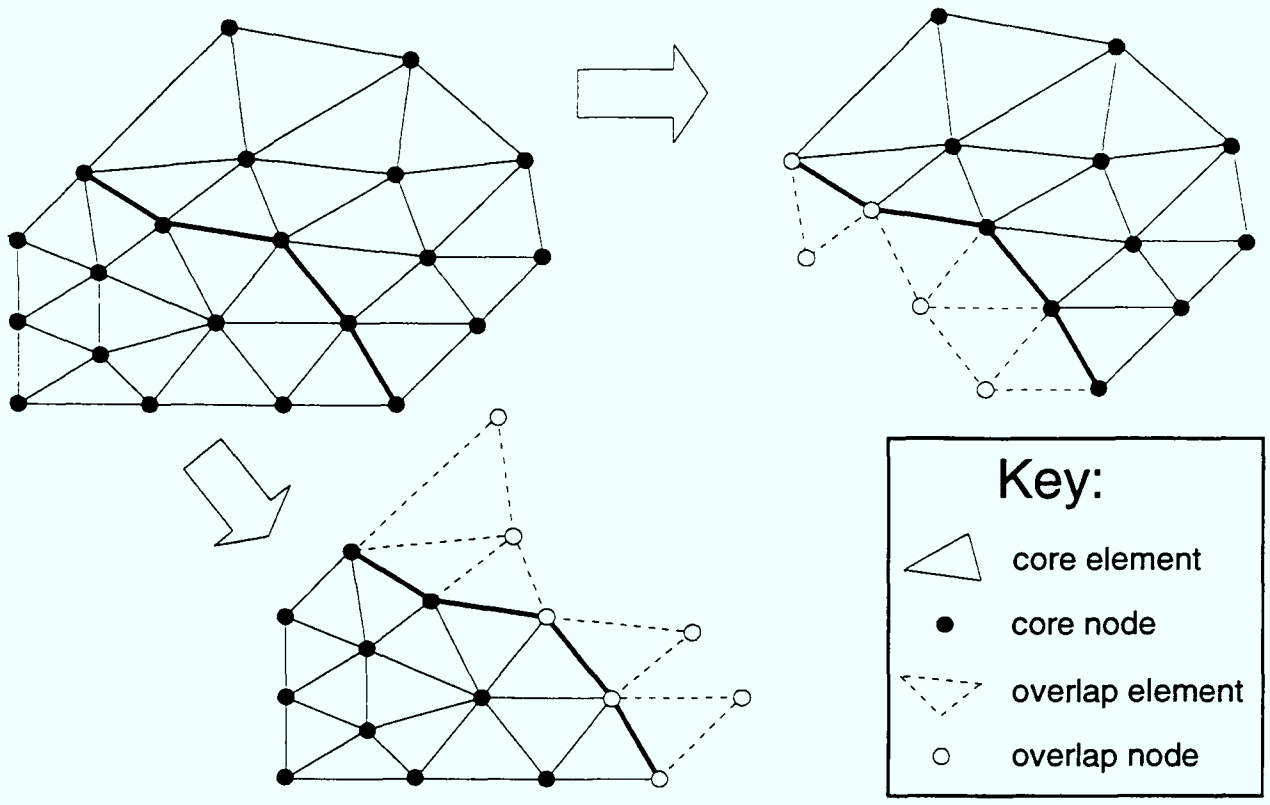

Figure 3.7: A mesh of 28 triangles divided into two sub-domains with the overlaps required for the stress scheme.

is implemented in the serial code. For example the array of grid points in Fortran may be declared as;

```
INTEGER DIMENSION, NO_OF_GRID_POINTS
INTEGER GRID_POINTS(1:DIMENSION, 1:NO_OF_GRID_POINTS)..
```

This array may be easily extended to include overlaps as;

```
INTEGER DIMENSION, EXTD_NO_OF_GRID_POINTS
INTEGER GRID_POINTS(1:DIMENSION, 1:EXTD_NO_OF_GRID_POINTS)
```

Clearly this structure will still be correctly declared in all subsequent subroutines calls without any code modification. Subroutines may be called with either the original or the extended point count and the declaration will remain consistent. If however the array of grid points is declared as;

```
INTEGER GRID_POINTS(1:NO_OF_GRID_POINTS, 1:DIMENSION)
```

Then the array may also be extended as;

```
INTEGER GRID_POINTS(1:EXTD_NO_OF_GRID_POINTS, 1:DIMENSION)
```

But now each subroutine must declare grid points to the extended size in order to remain consistent. It may prove less invasive to change the serial code to reverse such declarations and subsequently all occurrences of the variable. Apart from cache effects such a modification will not affect the serial code and unlikely to raise objections from the serial code authors.

3.3.3 Parallel Execution Control and Renumbering

Consider the following code fragment that loops over each grid point in each element.

```
INTEGER   NUMBER_OF_GP_IN_ELEMENT(1:NUMBER_OF_ELEMENTS)
INTEGER   GP_IN_ELEMENT(1:MAX_NUM_GP_PER_ELE,1:NUMBER_OF_ELEMENTS)
REAL      XELE(1:NUMBER_OF_ELEMENTS)
REAL      YGP(1:NUMBER_OF_GRID_POINTS)

DO I = 1, NUMBER_OF_ELEMENTS
  DO J = 1, NUMBER_OF_GP_IN_ELEMENT(I)
    XELE(I) = XELE(I) + YGP(GP_IN_ELEMENT(J,I))
  END DO
END DO
```

Two arrays are used in this example to describe the element topology;

`NUMBER_OF_GP_IN_ELEMENT` is a vector that contains the number of grid points that are in each element.

`GP_IN_ELEMENT` is a two dimensional array that contains the grid point number for each grid point in each element.

Two data items are involved; an element based variable `XELE` and a grid point based variable `YGP` . This code fragment can be implemented in parallel by using 'owner computes' execution control masks which are conditionals to control the scope of operations

for each processor. In this example the execution control mask is implemented with a function OWNER_OF_ELEMENT that returns true only if the argument is an element number that is owned by the processor, the computation only being performed if this is the case.

```
DO I = 1, NUMBER_OF_ELEMENTS
  IF ( OWNER_OF_ELEMENT(I) ) THEN
    DO J = 1, NUMBER_OF_GP_IN_ELEMENT(I))
      XELE(I) = XELE(I) + YGP(GP_IN_ELEMENT(J,I))
    END DO
  END IF
END DO
```

However in order to achieve scalability of memory each processor can store only its own sub-domain. In this example the most fundamental mesh entity, the grid point, described as a set of coordinates, will renumber itself through the simple process of being packed into memory as a consecutive list of coordinates for each grid point in the sub-domain. So the core grid points are packed into the first 1 to LOCAL_NUMBER_OF_GRID_POINTS locations and the overlap grid points as LOCAL_NUMBER_OF_GRID_POINTS+1 to EXT_LOC_NUMBER_OF_GRID_POINTS. Where LOCAL_NUMBER_OF_GRID_POINTS is the number of grid points in the sub-domain core and EXT_LOC_NUM_OF_GRID_POINTS is the number of grid points in the entire sub-domain. Similarly extracting and storing (packing) only the local entries for the variables XELE, YGP and NUMBER_OF_GP_IN_ELEMENT is straightforward. Other mesh entities are however described as relationships or 'pointers' between entities. So packing GP_IN_ELEMENT results in a list of global node numbers for each locally numbered element. To allow for this pointer arrays must be embedded into the code in order that each time the code refers to a grid point of an element the pointer array indirectly addresses a grid point in the local numbering scheme.

```
INTEGER   NUMBER_OF_GP_IN_ELEMENT(1:EXT_LOC_NUM_OF_ELEMENTS)
INTEGER   GP_IN_ELEMENT(1:MAX_NUM_GP_PER_ELE,1:EXT_LOC_NUM_OF_ELEMENTS)
INTEGER   PTR_ELE(1:NUMBER_OF_ELEMENTS)
INTEGER   PTR_GP(1:NUMBER_OF_GRID_POINTS)
REAL      XELE(1:EXT_LOC_NUM_OF_ELEMENTS)
REAL      YGP(1:EXT_LOC_NUM_OF_GRID_POINTS)

DO I = 1, NUMBER_OF_ELEMENTS
```

```
    IF ( OWNER_OF_ELEMENT(I) ) THEN
      DO J = 1, NUMBER_OF_GP_IN_ELEMENT(PTR_ELE(I))
        XELE(PTR_ELE(I)) = XELE(PTR_ELE(I)) +
+                            YGP(PTR_GP(GP_IN_ELEMENT(J,PTR_ELE(I))))
      END DO
    END IF
  END DO
```

Here two indirection pointer arrays are used PTR_ELE and PTR_GP which store the local element and grid point numbers respectively. For example if element number 28 is local element number 14 then PTR_ELE(28) has the value 14. The code still uses global numbers, only the addresses are indirected. A simple optimisation here is to move the element indirection upwards.

```
DO II = 1, NUMBER_OF_ELEMENTS
  IF ( OWNER_OF_ELEMENT(II) ) THEN
    I = PTR_ELE(II)
    DO J = 1, NUMBER_OF_GP_IN_ELEMENT(I)
      XELE(I) = XELE(I) + YGP(PTR_GP(GP_IN_ELEMENT(J,I)))
    END DO
  END IF
END DO
```

These pointers will need to be globally sized and so do not scale in memory. Also the loop still increments over the global number of elements and so does not scale in processing. Execution of the control mask for every element can be a significant operation. Since PTR_ELE now represents the local renumbering implied by the array packing, the local element numbers in the above loop when the execution control mask is true will run from 1 to LOCAL_NUMBER_OF_ELEMENTS. Therefore a further optimisation is possible by changing the loop limits to local numbering.

```
DO I = 1, LOCAL_NUMBER_OF_ELEMENTS
  DO J = 1, NUMBER_OF_GP_IN_ELEMENT(I)
     XELE(I) = XELE(I) + YGP(PTR_GP(GP_IN_ELEMENT(J,I)))
  END DO
END DO
```

Now only one pointer is required but it remains globally sized and so is still not scalable. If all uses of GP_IN_ELEMENT throughout the code are as the index of the array PTR_GP

then this indirection can be propagated upwards to the highest level where PTR_GP is used to renumber the contents of GP_IN_ELEMENT to a local grid point numbering scheme. The example now becomes

```
DO I = 1, LOCAL_NUMBER_OF_ELEMENTS
  DO J = 1, NUMBER_OF_GP_IN_ELEMENT(I)
     XELE(I) = XELE(I) + YGP(GP_IN_ELEMENT(J,I))
  END DO
END DO
```

If this code fragment exists inside a subroutine where NUMBER_OF_ELEMENTS is passed into the subroutine as an argument then the calling routine can be modified to call the subroutine with LOCAL_NUMBER_OF_ELEMENTS so that *no code modification is required in the subroutine.*

This thesis follows the option of re-numbering each entire sub-domain to a local numbering scheme as this has been shown above to be consistent with objectives (ii) and (iii). Each processor 'sees' its renumbered sub-domain as a complete mesh consisting of 1 to n_e elements and 1 to n_p grid points where n_e and n_p are the local number of elements and grid points respectively. This can be carried out at the highest possible level in the code, that is where the problem specification is read from file. A record of the global (serial) numbers for each local mesh entity (referred to as a decomposition index) is stored on each processor in order to allow reconstruction of data back into its original global form. Translation back from local to global numbering using this record is only required as an i/o process when writing variables to file. Rebuilding of global variables is carried out by the i/o (master) processor and so this is the only processor that requires the decomposition indices, however the indices are distributed with the sub-domains to maintain scalability of memory. This scheme can encounter difficulty when the problem size increases to the point at which the geometry description will no longer fit into the memory of the master processor. This is not however insurmountable and is discussed further in Section 4.2 and Chapter 7. The effect of renumbering is illustrated in Figures 3.8 and 3.9. Consider the element partition in Figure 3.9 The partition list P_e of processor numbers that own each element as returned from the partitioner utility is as follows;

```
1 1 1 1 1 1 2 2 2 2 2 2 2 1 1 1 1 1 1 1 2 2 2 2 2 2 2
```

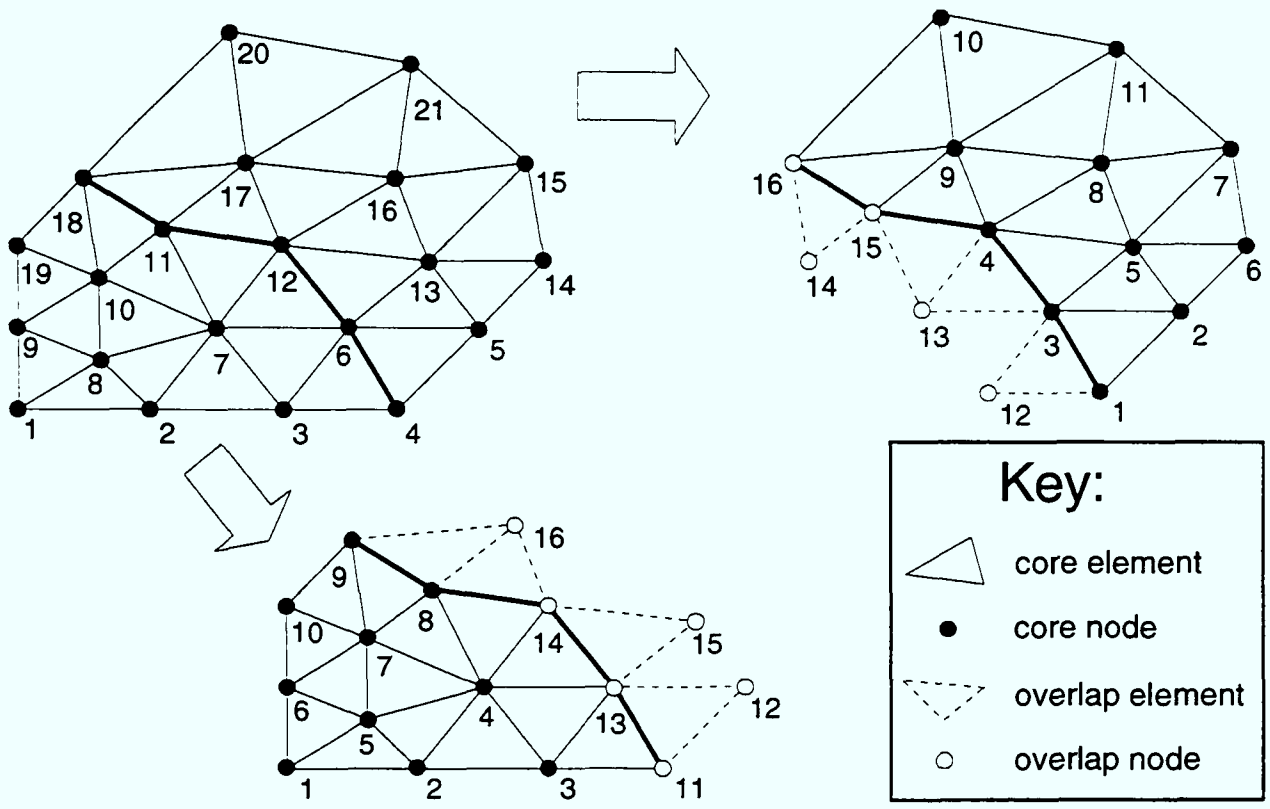

Figure 3.8: A mesh of 28 triangles divided into two sub-domains showing the renumbering of grid points from global to local numbering.

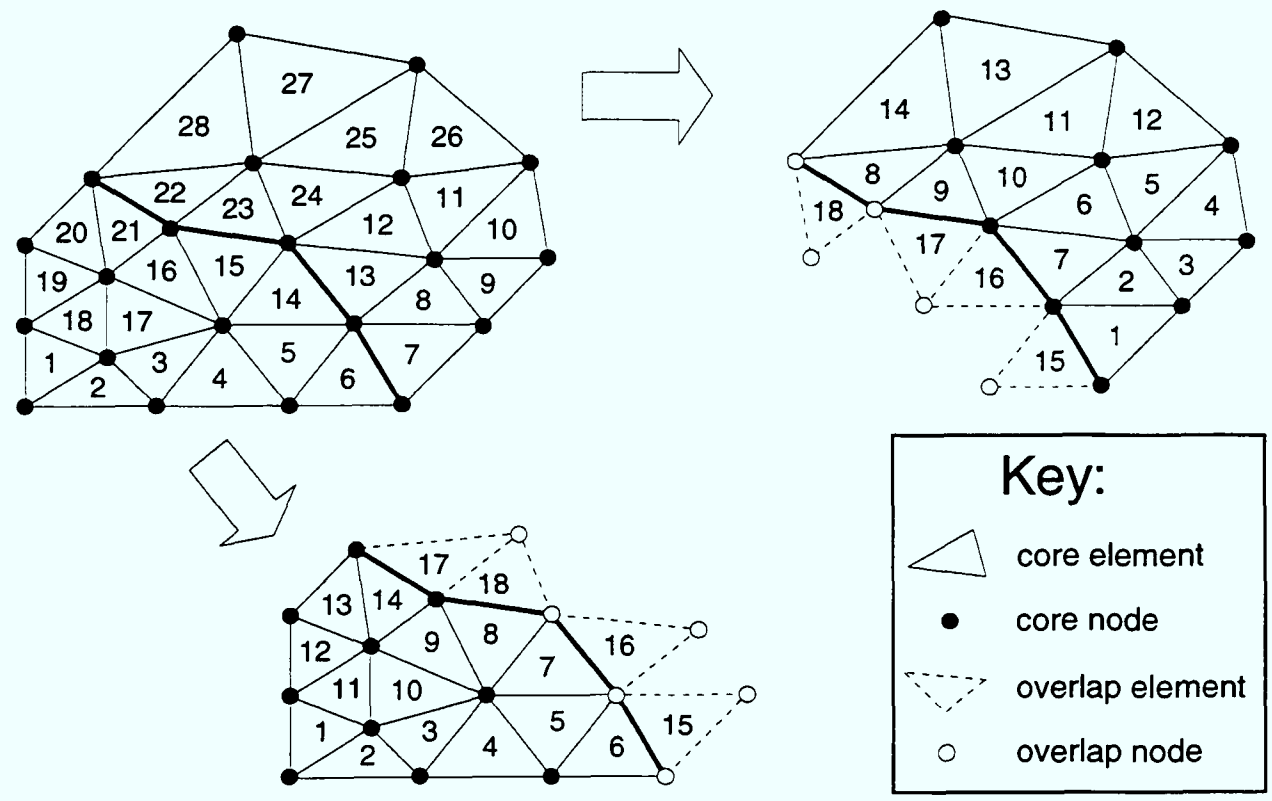

Figure 3.9: A mesh of 28 triangles divided into two sub-domains showing the renumbering of elements from global to local numbering.

The resulting element renumbering as stored in PTR_ELE is listed in Table 3.2. The

Global	Processor1	Processor2
1	1	0
2	2	0
3	3	0
4	4	0
5	5	0
6	6	15
7	15	1
8	0	2
9	0	3
10	0	4
11	0	5
12	0	6
13	16	7
14	7	16
15	8	17
16	9	0
17	10	0
18	11	0
19	12	0
20	13	0
21	14	18
22	17	8
23	18	9
24	0	10
25	0	11
26	0	12
27	0	13
28	0	14

Table 3.2: Element indirection pointer arrays for the partition illustrated in Figure 3.9

renumbering has maintained the core elements as the first 14 elements in each partition allowing the transformation to local loop limits. The implications of renumbering are discussed further in Section 4.3.

3.3.4 Overlap Communication

The notion of the mesh overlaps is that each processor calculates only the values of core variables. That is variables that are associated with mesh entities within its own domain,

mitted. A corresponding template records the entity numbers to be received and the processor number from which they will arrive. These templates must be matched across each sub-domain boundary so that the data sent from one sub-domain is received in the anticipated order in the adjacent sub-domain. This is achieved by preserving the global ordering of the elements. For a simple processor interconnection topology such as a pipeline (a one dimensional chain), where the partition can guarantee mapping to the processor topology, the template becomes reasonably straightforward. Exchange of data between processors can be synchronised by the template on an odd-even alternate pair basis. This is a four cycle process described in the following table.

Processor Number	Odd	Even
	Send right	Receive left
	Receive right	Send left
	Send left	Receive right
	Receive left	Send right

Table 3.3: Communication operations required for a simple chain of processors

This simple scheme enables the exchange to be carried out as a parallel process. More elaborate processor topologies can be handled with variations on such a scheme. Regular two dimensional processor arrays can for instance use red - black checkerboard type schemes. It cannot however be assumed that the mesh can be partitioned in such a way as to map perfectly to the processor interconnection topology (Section 3.2.3). A scheme is required which can cope efficiently with an unstructured partition of an unstructured mesh mapped imperfectly to an array of processors. This is a scheduling problem of the type familiar to operational research [Wil84].

The scheme adopted involves constructing the graph $G(P, C)$ of processors P and sub-domain (processor) interconnections C and attaching weights to the interconnects according to the size of the interface. This graph is initially sorted by weight with the processor pair having the largest amount of data to communicate being first. The graph is then scheduled to provide a sequence in which exchanges occur as a parallel process

with the largest exchanges first. Starting with the heaviest node pair, the processor numbers are recorded. The graph is then searched for the next heaviest weight that does not use one of the already recorded processors. When found this processor pair is sorted to be the next entry in the graph. This operation is carried out until either all processors are involved in communication or an unrecorded processor pair is no longer available for scheduling. If there are still entries in the graph that have not been scheduled the list of recorded processors is cleared and the process repeated until all processor pairs have been scheduled. This results in a layering of exchange communication processes which should be (but is not guaranteed to be) no deeper than the maximum node degree of the processor graph $G(P, C)$.

Consider the mesh illustrated in Figure 3.11 decomposed into three renumbered subdomains in Figure 3.12 Here the overlap renumbering has followed the original global

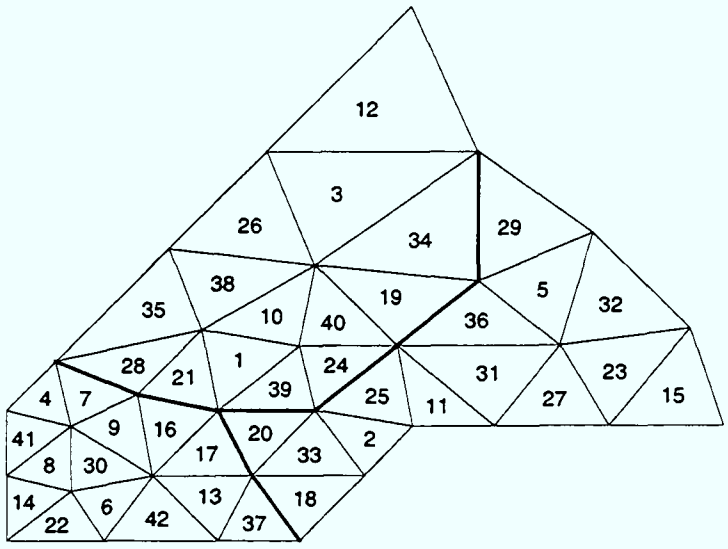

Figure 3.11: Mesh of 42 triangular elements.

numbering scheme Processor (a) must receive data for overlap elements 17 and 18 from processor (b) where they are numbered 6 and 9 respectively. Similarly processor (b) must receive data for overlap elements 15 and 16 from processor (a) where they are numbered 3 and 8 respectively. The communications for this example may be carried out in six stages as follows:

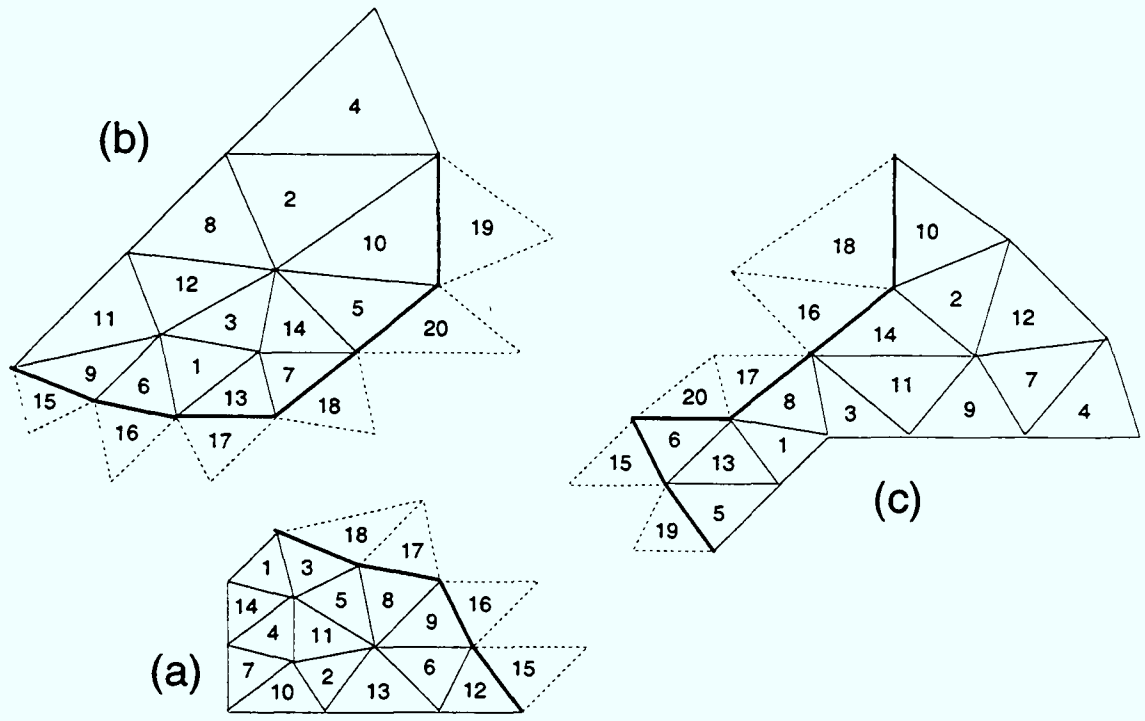

Figure 3.12: Mesh of 42 triangular elements partitioned into three renumbered subdomains.

Processor (a)

1 Sending to processor (b) elements 3 and 8

2 Receiving from processor (b) elements 17 and 18

3 Sending to processor (c) elements 9 and 12

4 Receiving from processor (c) elements 15 and 16

Processor (b)

1 Receiving from processor (a) elements 15 and 16

2 Sending to processor (a) elements 6 and 9

5 Sending to processor (c) elements 5, 7, 10, and 13

6 Receiving from processor (c) elements 17, 18, 19 and 20

Processor (c)

3 Receiving from processor (a) elements 15 and 19

4 Sending to processor (a) elements 5 and 6

5 Receiving from processor (b) elements 16, 17, 18 and 20

6 Sending to processor (b) elements 6, 8, 10 and 14

Note that these element numbers are always in increasing order both globally and locally. The sending is always carried out first to allow parallelism in packing.

Data that is to be transmitted from a sub-domain core is collected into a data buffer which allows one transmission and therefore only one latency to complete the transfer. Unpacking of data from a buffer is an overhead that is not necessary for data reception. Data is only ever received into an overlap, so arranging for the overlap renumbering scheme to consecutively number overlap entities that are owned by the same processor allows incoming data to be received directly into the overlap memory. So the global number ordering is preserved for each interface to other processors, but not throughout the overlap. In the above example elements 15 and 19 on processor (c) are in the core of processor (a) and so should be numbered consecutively. This involves renumbering overlap element 19 on processor (c) to be 16 and then overlap elements 16, 17, 18 and 20 to be 17, 18, 19 and 20 respectively.

Chapter 4

Algorithm Decomposition

The algorithms employed in unstructured mesh codes have invariably been developed using the traditional Von Neumann programming model of sequential instruction execution. The conversion of these serial algorithms into parallel algorithms may be straightforward, or may be very involved. Parallelism exists in many forms with a CM code. Having chosen a geometric (topologic) DD strategy, decomposition of the algorithms to concurrently operate locally within each sub-domain whilst performing the same operations as the original serial algorithm becomes a largely automatic process of communicating data as and when required. Profiling CM execution shows that the majority of run time is spent within the matrix equation solvers. It is these solvers that are subjected to close scrutiny to extract the maximum possible parallel efficiency Ideally we should be able to meet objective (i) and produce results from the parallel code that identically match the results produced by the serial code. This may not however, be either practical or possible. A variation between the serial and parallel code is sometimes inevitable. There are instances where it may be more important for example to meet objective (iii) and produce a highly efficient parallel code at the expense of failing to precisely meet objective (i). Again it will usually be a case of having to make an intelligent decision as to which is the overridingly important criteria. Often there is little choice but to either modify the algorithm or else suffer unacceptable inefficiency.

4.1 UIFS - Unstructured Incompressible Flow and Stress

The code used as a vehicle for developing the parallel strategies used in this thesis is known as UIFS. Developed for the purpose of modelling the processes involved in metals casting UIFS is a 2D unstructured mesh code for solving the Navier Stokes equations for transient and steady state flow problems with solidification [Cho93] along with the elastic stress-strain equations [FBCL91, CBCP92]. The techniques developed for UIFS have led to the development of the 3D code PHYSICA which provides even greater modelling flexibility for multi-physical processes.

4.1.1 The FV Fluid Dynamics Scheme

The Finite Volume (FV) (irregular control volume) fluid dynamics scheme in UIFS solves for flow on a single unstructured mesh using a modification of the SIMPLE algorithm of Patanker and Spalding [Pat80]. This is a cell centred scheme in which the control volume is formed by the elements of the mesh which may be any arbitrary shape. The definition of a staggered grid as used by Patanker *et.al.* is not clear for an unstructured mesh. So the scheme uses a co-located grid with the Rhie and Chow [RC82] pressure weighted interpolation method to suppress pressure oscillation. The solidification scheme uses the Voller and Cross enthalpy method [VCM87] to model the velocity correction and latent heat release during phase change. The dependency required by the solvers in this element centred finite volume scheme is simple nearest neighbour as illustrated in Figure 1.3(a). However in order to evaluate the cell volumes for the displaced grid the grid point dependency as shown in Figure 1.3(d) is also required. Hence the definition of the overlap mesh entities as given in Section 3.3.2. The scheme produces a sparse irregular diagonally dominant system matrix which may be solved using either Jacobi or Gauss Seidel SOR iterative methods. The fluid dynamics loop is illustrated in Figure 4.3. The number of iterations for each of the momentum, pressure and heat solvers are set at run time along with the maximum and minimum number of sweeps around the fluid dynamics loop. Convergence is based on the residuals of all of momentum, heat and

pressure variables.

Momentum Equations

The equations governing the conservation of momentum for an incompressible fluid in a cartesian system of coordinates may be expressed as:

$$\frac{\partial(\rho u_i)}{\partial t} + \nabla \cdot (\rho \underline{v} u_i) = \nabla \cdot (\mu \nabla u_i) - \frac{\partial p}{\partial n_i} + s_{u_i} \tag{4.1}$$

Here u_i is the momentum in the i axis, similar equations govern the momentum in the other axis. The other terms are; the density ρ the resultant velocity, $\underline{v}$, the viscosity μ, the pressure p, the face normal component n_i and the momentum source in the i axis s_{u_i}. The momentum source term includes the buoyancy source s_{b_i} and the Darcy source s_{d_i} terms which couple the momentum equation to the energy equation.

$$s_{u_i} = s_{b_i} + s_{d_i} + s_{\text{boundary}} + s_{\text{other}} \tag{4.2}$$

Continuity Equation

Then continuity equation governing mass conservation can be expressed as:

$$\frac{\partial \rho}{\partial t} + \nabla \cdot (\rho \underline{v}) = s_c \tag{4.3}$$

Here s_c is the mass source.

Energy Equation

Conservation of energy can be written as:

$$\frac{\partial \rho h}{\partial t} + \nabla \cdot (\rho \underline{v} h) = \nabla \cdot (k \nabla T) + s_h \tag{4.4}$$

Where h is the specific enthalpy, k is the thermal conductivity, T is the temperature and s_h is the volumetric source for heat. This equation may be expressed solely in terms of temperature using $h = cT$ where c is the specific heat.

Buoyancy Source

The source terms s_{u_i} in Equation 4.1 couples into the energy equation through the buoyancy terms. Two alternative buoyancy terms are available in UIFS; constant and variable density. The constant density approximation Boussinesq source s_{b_i} in the i direction can be expressed as

$$s_{b_i} = -\rho_{\text{ref}} \beta g_i (T - T_{\text{ref}}) \tag{4.5}$$

Where ρ_{ref} is the constant density, β is the volumetric coefficient of thermal expansion, T is the temperature, T_{ref} is the reference temperature (temperature for ρ_{ref}) and g_i is the acceleration due to gravity in the i direction. Density may be more accurately expressed as a function of temperature $\rho(T)$ so the buoyancy source becomes

$$s_{b_i} = \rho(T) g_i \tag{4.6}$$

Solidification Sources

For a system undergoing a change of phase from liquid to solid (or solid to liquid) the total enthalpy H can be expressed as the sum of the 'sensible' enthalpy h and the latent heat ΔH

$$H = h + \Delta H \tag{4.7}$$

Latent heat will be some function F of temperature

$$\Delta H = F(T) \tag{4.8}$$

which may be written in terms of the latent heat of solidification L and liquid fraction (ratio of liquid to solid) f_l

$$F(T) = L f_l \tag{4.9}$$

Combining this with Equation 4.4 gives the enthalpy source due to the latent heat of solidification as

$$s_h = -\frac{\partial \rho L f_L}{\partial t} - \nabla \cdot (\rho \underline{v} L f_L) \tag{4.10}$$

Velocity correction for changes in material properties during phase transition uses the Darcy source term

$$s_{d_i} = -\frac{\mu}{K} u_i \tag{4.11}$$

where μ is the viscosity and K is the permeability. Little data is available for the viscosity and permeabilities of materials undergoing phase transition so a simple approximation involving the liquid fraction is used

$$s_{d_i} = -B(1 - f_l)u_i \tag{4.12}$$

where B is an empirical constant.

4.1.2 The FV Solid Mechanics Scheme

The grid point (vertex) based solid mechanics code uses the finite volume unstructured mesh procedure of Fryer *et.al.* [FBCL91, Fry93] for the solution of the elastic stress-strain equations for bodies undergoing thermal or mechanical loads.

Governing Equations

The general equilibrium equations governing the conservation of force on a static body are

$$\begin{aligned} \frac{\partial \sigma_{xx}}{\partial x} + \frac{\partial \sigma_{xy}}{\partial y} &= f_x \\ \frac{\partial \sigma_{yy}}{\partial y} + \frac{\partial \sigma_{xy}}{\partial x} &= f_y \end{aligned} \tag{4.13}$$

Where σ_{ii}, σ_{ij} and f_i are the components of normal stress, shear stress and body forces acting in direction i. In matrix form the above equations become

$$\boldsymbol{\sigma} = \mathbf{D}\boldsymbol{\varepsilon}^{(\mathrm{e})} \tag{4.14}$$

where the stress vector is $\boldsymbol{\sigma} = (\sigma_{xx} \;\; \sigma_{yy} \;\; \sigma_{xy})^{\mathrm{T}}$ and the elastic strains are $\boldsymbol{\varepsilon}^{(\mathrm{e})} = (\varepsilon_{xx}^{(\mathrm{e})} \;\; \varepsilon_{yy}^{(\mathrm{e})} \;\; \varepsilon_{xy}^{(\mathrm{e})})^{\mathrm{T}}$. The matrix $\boldsymbol{D}$ holds the material elastic properties; Youngs modulus

E and Poissons ratio μ where for plane strain

$$\mathbf{D} = \frac{E}{(1-\mu^2)} \begin{bmatrix} 1 & \mu & 0 \\ \mu & 1 & 0 \\ 0 & 0 & \frac{1-\mu}{2} \end{bmatrix} \tag{4.15}$$

The total strain $\boldsymbol{\varepsilon}^{(\mathrm{T})}$ is related to displacement by

$$\boldsymbol{\varepsilon}^{(\mathrm{T})} = \mathbf{L}\mathbf{d} \tag{4.16}$$

Where the displacement vector $\mathbf{d} = (u\ v)^{\mathrm{T}}$ represents displacement in the x and y directions and $\mathbf{L}$ holds the differential operators

$$\mathbf{L} = \begin{bmatrix} \frac{\partial}{\partial x} & 0 \\ 0 & \frac{\partial}{\partial y} \\ \frac{\partial}{\partial y} & \frac{\partial}{\partial x} \end{bmatrix} \tag{4.17}$$

Thermal strains are given by

$$\boldsymbol{\varepsilon}^{(\mathrm{Th})} = (1+\mu)\alpha\Delta T\mathbf{i} \tag{4.18}$$

where α is the coefficient of thermal expansion, ΔT is the temperature change and $\mathbf{i} = (1\ 1\ 0)^{\mathrm{T}}$.

Discretisation of the Solution Domain

This scheme forms a control volume around each grid point with contributions to the control volume from each of the surrounding mesh elements as illustrated in Figure 4.1. Here the sub control volumes in each surrounding element are formed by connecting the element centres to the face centres. Temperature and displacement variables are stored at the grid points and the material properties, Youngs modulus, Poisson ratio, etc., are associated with the elements. The equilibrium equations are integrated over the control volumes where the divergence theorem is used to transform the area integrals

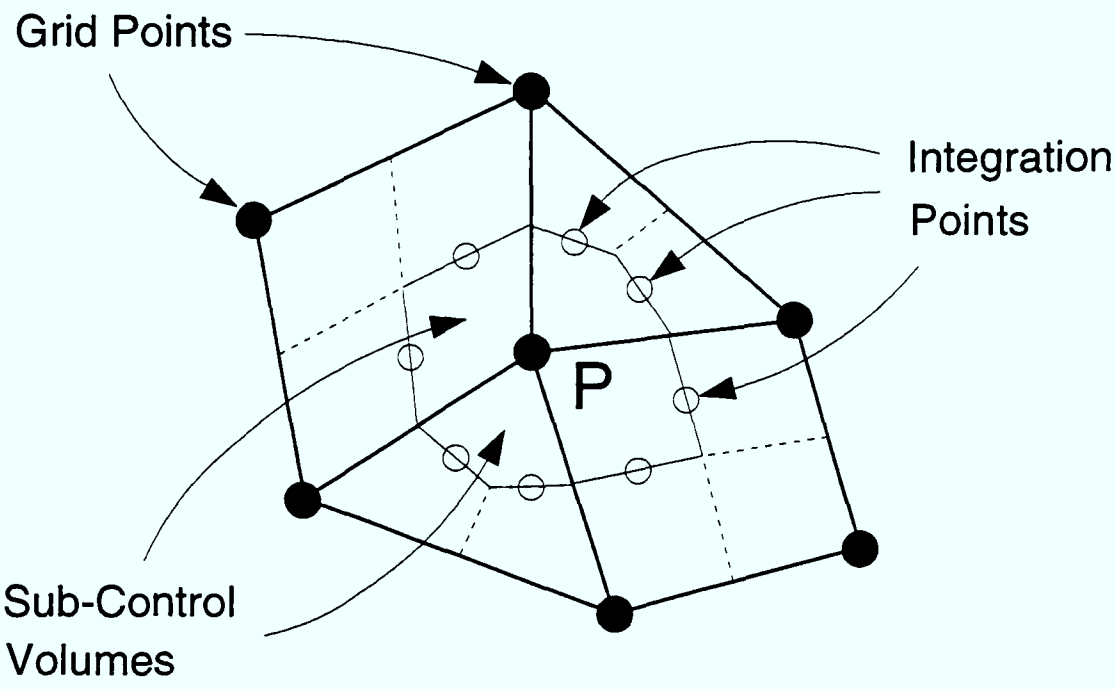

Figure 4.1: Formation of a control volume from sub-control volumes around point P.

into line integrals which enables the stresses to be approximated at the integration points on the surface of the control volume. The discretisation uses reference elements to represent the mesh elements in a local coordinate system in a manner similar to the Finite Element (FE) method [SR87] (Figure 4.2). This is a computationally efficient scheme which obtains approximations to the derivatives in the equilibrium equations in local coordinates and uses a Jacobian matrix to map the approximations back to global coordinates. A variable ϕ and its derivatives can be approximated anywhere within an element of m grid points using Equations 4.19 and 4.20.

$$\phi(s,t) = \sum_{i=1}^{m} N_i(s,t)\phi_i = N\phi \tag{4.19}$$

$$\frac{\partial \phi(s,t)}{\partial k} = \sum_{i=1}^{m} \frac{\partial N_i(s,t)}{\partial k}\phi_i \tag{4.20}$$

$$k = s,t$$

The shape functions N_i for a bilinear quadrilateral are

$$N_1(s,t) = 0.25(1+s)(1+t)$$

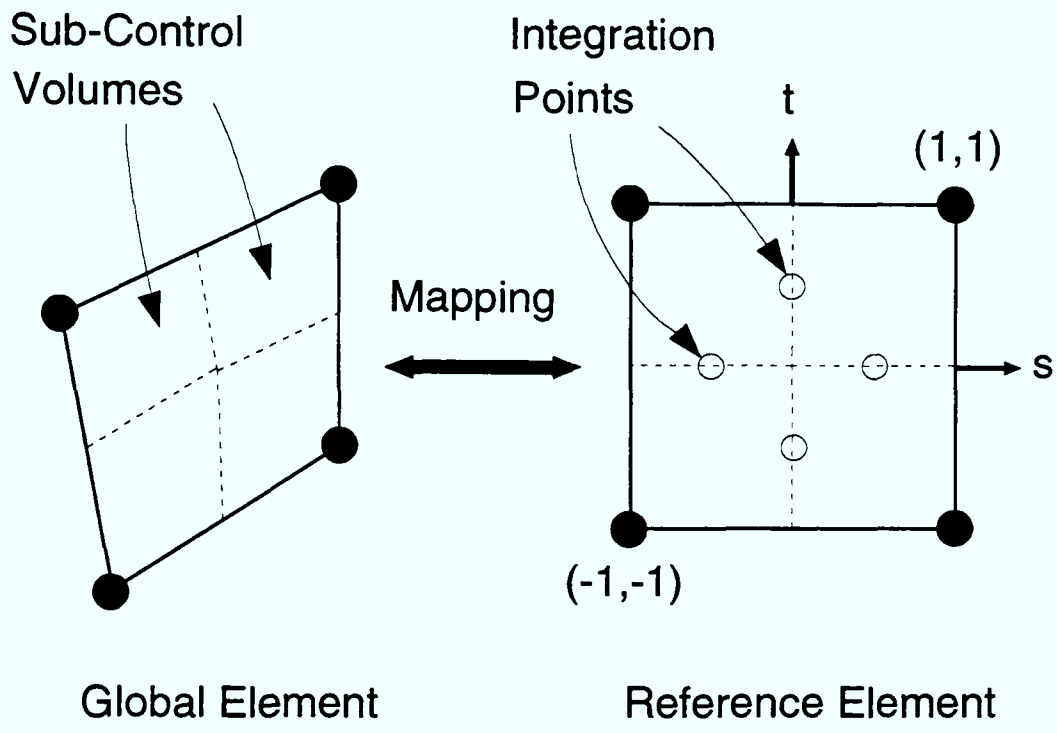

Figure 4.2: Mapping of a finite volume element to a reference element.

$$N_2(s,t) = 0.25(1-s)(1+t)$$

$$N_3(s,t) = 0.25(1-s)(1-t)$$

$$N_4(s,t) = 0.25(1+s)(1-t)$$

The Jacobian matrix in Equation 4.21 is used to map the derivatives of the shape function from local to global coordinates.

$$\begin{bmatrix} \frac{\partial N_i}{\partial x} \\ \frac{\partial N_i}{\partial y} \end{bmatrix} = \begin{bmatrix} \frac{\partial x}{\partial s} & \frac{\partial y}{\partial s} \\ \frac{\partial x}{\partial t} & \frac{\partial y}{\partial t} \end{bmatrix}^{-1} \begin{bmatrix} \frac{\partial N_i}{\partial s} \\ \frac{\partial N_i}{\partial t} \end{bmatrix} \tag{4.21}$$

$$\frac{\partial x}{\partial s} = \sum_{i=1}^{m} \frac{\partial N_i}{\partial s} x_i$$

$$\frac{\partial x}{\partial t} = \sum_{i=1}^{m} \frac{\partial N_i}{\partial t} x_i$$

$$\frac{\partial y}{\partial s} = \sum_{i=1}^{m} \frac{\partial N_i}{\partial s} y_i$$

$$\frac{\partial y}{\partial t} = \sum_{i=1}^{m} \frac{\partial N_i}{\partial t} y_i$$

Where x_i and y_i are the grid point coordinates of the element.

Discretisation of the equilibrium equations

The tensor form of the equilibrium equation is

$$\frac{\partial \sigma_{ij}}{\partial x_j} = f_i \tag{4.22}$$

Integrating over a control volume

$$\int_\Omega \left(\frac{\partial \sigma_{ij}}{\partial x_j} - f_i \right) d\Omega = 0.0 \tag{4.23}$$

Using the divergence theorem

$$\int_\Omega \frac{\partial \sigma_{ij}}{\partial x_j} \, d\Omega = \int_S \sigma_{ij} \cdot n_j \, dS \tag{4.24}$$

where S is the surface of the control volume. Which gives the matrix form for the integral over the surface of the control volume

$$\oint_S \boldsymbol{\sigma} \cdot \mathbf{n} \, dS = \int_\Omega \mathbf{f} \, d\Omega \tag{4.25}$$

Substituting the stress-displacement relationship $\boldsymbol{\sigma} = \mathbf{DLNu} - \mathbf{D}\boldsymbol{\epsilon}^{(\mathrm{Th})}$ with $\mathbf{B} = \mathbf{LN}$ into Equation 4.25 gives an integral expression in terms of the nodal displacements $\mathbf{u}$ for each control volume.

$$\oint_S (\mathbf{DBu} - \mathbf{D}\boldsymbol{\epsilon}^{(\mathrm{Th})}) \cdot \mathbf{n} \, dS = \int_\Omega \mathbf{f} \, d\Omega \tag{4.26}$$

rearranging to give displacements in terms of strain

$$\oint_S (\mathbf{DBu}) \cdot \mathbf{n} \, dS = \int_\Omega \mathbf{f} d\Omega + \oint_S \mathbf{D}\boldsymbol{\epsilon}^{(\mathrm{Th})} \cdot \mathbf{n} \, dS \tag{4.27}$$

For plane stress the stress-displacement relationships are

$$\begin{aligned}
\sigma_{xx} &= \frac{E}{(1-\mu^2)} \left[\frac{\partial u}{\partial x} + \mu \frac{\partial v}{\partial y} - (1+\mu)\alpha T \right] \\
\sigma_{yy} &= \frac{E}{(1-\mu^2)} \left[\frac{\partial v}{\partial y} + \mu \frac{\partial u}{\partial x} - (1+\mu)\alpha T \right] \\
\sigma_{xy} &= \frac{E}{2(1+\mu)} \left[\frac{\partial u}{\partial y} + \frac{\partial v}{\partial x} \right]
\end{aligned} \tag{4.28}$$

Boundary Conditions

For control volumes at the boundary of the domain Γ the contribution of the faces that lie on the boundary are given as boundary conditions.

$$-\int_{\Gamma} \mathbf{DBu} \cdot \mathbf{n}\, d\Gamma \tag{4.29}$$

This surface integral can represent displacements and loads applied to the domain surface.

Solution Procedure

For each axis, coefficients for each node are assembled to form a sparse irregular diagonally dominant system matrix $\mathbf{A}$. The vector $\mathbf{x} = (u_1, \cdots u_n)$ where n is the number of nodes in the mesh represents the displacements for this axis. The vector $\mathbf{b}$ represents the source terms from the temperature changes, stresses and boundary conditions. The equation $\mathbf{Ax} = \mathbf{b}$ is solved using the diagonally preconditioned conjugate gradient method.

4.1.3 Integration within UIFS

The fluid mechanics code is loosely coupled with the solid mechanics code as shown in Figure 4.3. Here the fluid dynamics loop reaches convergence for a time step before entering the solid mechanics loop. When the solid mechanics loop reaches convergence UIFS loops for the next time step. Each of the solvers may be turned on or off to suit the requirements of a given problem. As the fluid mechanics stage often requires more effort to obtain a satisfactory solution than solid mechanics, the elastic solver loop may be masked to only run every kth time step. Even with $k = 1$ the bulk of the computational effort is usually expended in the fluid mechanics loop. This is of course problem dependent, for a solidification type problem the initial time steps may be entirely fluid and the closing time steps entirely solid.

Discretisation of the integration of the governing equations leads to matrix equations that exhibit localised dependencies across the mesh. Solution of an element requires

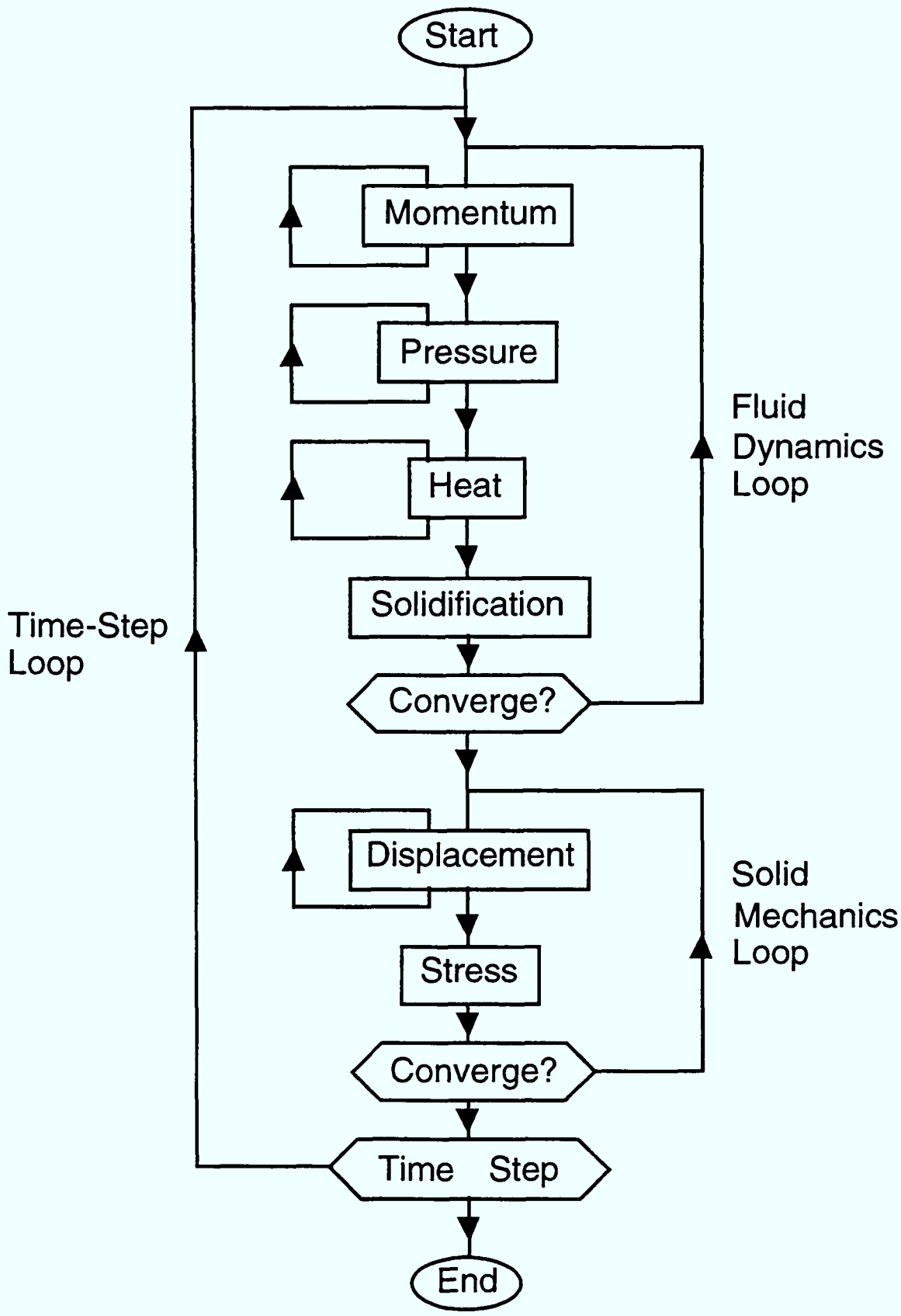

Figure 4.3: Flowchart for UIFS.

data from its neighbouring elements. From the perspective of parallelisation the details of the solution schemes are important only in so far as they give a description of the data dependency. The way in which neighbouring variables and related variables interdepend in the solution system is the overriding concern for parallelisation. It is important to realise the close interaction of the variables, from the point of initialisation onwards the solution of any one variable is dependent upon many previously solved variables. Momentum is used to solve for pressure, which in turn is used to solve for energy, energy for solidification, solid fraction for displacement, displacement for momentum and so the cycle of dependence continues. This relationship places bounds on what and where to communicate. The use of data in an overlap indicates that a communication will be required prior to the calculation, this communication must be performed after the required data is calculated in a previous stage.

4.2 Parallelisation of UIFS

The bulk of the Parallel UIFS (PUIFS) code remains almost untouched by parallelisation. Following the SPMD paradigm each processor has a copy of the entire (PUIFS) code which is executed in a similar manner to the serial case. The parallelism is to a significant extent hidden 'behind the scenes' while the code runs. The point at which a UIFS problem is parallelised is as the problem geometry specification (grid points, element topology, element adjacency, boundaries) is read from file. This provides a clear interface between the serial problem as it exists on file and the parallel problem as it exists in the distributed memory. At run time the PUIFS code reads the problem in a decomposed form from file. That is a problem which has been partitioned into re-numbered sub-domains along with some extra data to specify the overlaps and communications. Decomposition of the problem files may be carried out transparently at run time on the i/o processor or executed as a preprocessing task on the problem files possibly using another machine. This allows processing of problems that are too large for the geometry to be accommodated by the memory of one node in the parallel machine. Also the

same decomposed problem may be re-run with altered boundary conditions or material properties without the need to pre-process each time. As each part of the problem is read in (by the one i/o processor (master)) the parts are distributed to the appropriate processor. The data space required to store the extra variables is concealed as a common block that is included into parallelised routines, this is given in Appendix A. This system is actually a master slave paradigm during the i/o process, there may be only one source code but it contains conditionals such as;

```
IF ( MASTER ) THEN
```

Once the decomposed problem has been loaded onto the processors each processor acts on its own sub-domain as if it were a self contained problem. Execution on each processor is synchronised in information exchanges in order that the global problem remains consistent. At the end of the run the results and re-start variables are dumped to file in exactly the same format as a serial code run. Reconstruction of the global variables from the decomposed variables is carried out by the i/o processor Each processor in sequence hands its variable back to the i/o processor along with the global numbering scheme required to place the variables into global order. The current implementation requires that the i/o processor has sufficient memory to allow the re-construction of a single global sized data item.

4.2.1 Partitioning

The JOSTLE program [WCE+95] is used to provide a partition of the mesh. JOSTLE operates on a graph that in the case of UIFS represents the mesh and returns a partition of that graph (Appendix B). For PUIFS the dual graph of the mesh is used to obtain a partition based on elements. The dual graph is the graph in which the nodes or vertices of the graph represent the elements of the mesh and the graph edges represent the element adjacency (connectivity). For the purposes of experimentation JOSTLE can be run as a stand alone program that produces a file describing the mesh partition. This allows for flexibility in adjusting the parameters used to control the partition and visualisation of the partition produced. JOSTLE has also been embedded into PUIFS

so that a partition may be produced rapidly at run time. The partition produced by JOSTLE (primary partition) is used to generate a secondary partition for the mesh grid points as described in Section 3.3.1. The primary and secondary partitions are inverted to generate lists of the global element and grid point numbers that exist in each sub-domain. The rules for overlap generation given in Section 3.3.2 are applied to produce descriptions of the overlaps in a global numbering scheme. The element and grid point lists are extended to contain the global element and grid point numbers for the overlaps. Boundaries in UIFS are described as a set of grid points and boundary conditions are described in file as a set of 'patches'. This allows the boundary points along with the associated boundary patch number to be partitioned in accordance with the extended grid point partition. The boundary patches and material properties are not partitioned. These parameters are read at run time and distributed to all processors whether or not they are needed on that processor. For small numbers of processors ($P < 500$) this is an insignificant memory overhead for PUIFS exchanged for simplicity of the code.

4.2.2 Renumbering

To create self-contained sub-domains the extended mesh partitions are renumbered into local numbering schemes as described in Section 3.3.3. All element and grid point based variables are packed and renumbered with overlaps following the core data. All loops in PUIFS are transformed into local loops, all mesh entity relationships (element topology, connectivity, etc.) are renumbered to local numbering along with the boundary grid point lists. Therefore no execution control masks or indirection pointer arrays are required.

4.2.3 Communication

Overlap communication schedules are calculated at the beginning of a code run as part of the decomposition process. This is calculated once using the global problem geometry as read from file. The decomposed problem definition is subsequently distributed along with the communication schedules to the appropriate processors. Any invariant quantities are communicated once only at the start of the code before entering the timestep loop.

All other variables are communicated as and when required using the communication schedules calculated at the start of the code.

4.2.4 Parallel Utilities

The parallel utility library developed for PUIFS is described in Appendix A. The routines `PARTITION`, `SECONDARY` and `DECOMPOSE` are used either at the start of the code run to decompose the problem into sub-domains or as components of a preprocessor to pre-partition the problem specification.

The key communication utility is `SWAP( VARIABLE, SPATIAL_REFERENCE )` which performs an exchange of overlap data for the input `VARIABLE` between all processors in accordance with the communication schedules. The `SPATIAL_REFERENCE` argument defines which of the communication schedules to be used, i.e. element or grid point. Overlap exchange is a highly parallel process which involves a matching send and receive operation across all sub-domain boundaries. The time required for a `SWAP` is approximated as $2s_{max}t_m$ where s_{max} is the maximum number node order in the processor graph $G(P, C)$ in Section 3.3.4 and t_m is the average time to send a message. The important point here is that the number of processors P does not feature highly in this approximation and so `SWAP` scales well, the time required being independent of P.

Global commutative (reduce) operations (`GSUM`, `GMAX`, `GAND`, etc.) are used to obtain global values of a commutative function by combining local partial evaluations of the function and broadcasting the results to each processor. The time required for a global commutative operation is dependent on the actual implementation of the operation which can vary with partition strategy, communication harness and platform hardware. For example a global commutative may be implemented on a chain of processors by passing all partial evaluations back to the master processor where the global value can be evaluated which is then passed back along the chain in a broadcast to all processors. The time required for this operation will consequently be something like to $2(P-1)t_l$ where t_l is the communication start up time (latency). With a mesh of $p \times q$ processors a similar strategy will require $2(p+q-2)t_l$. No matter how a global operation is implemented the

time required increases with increasing P and so does not scale well. Care is therefore required in avoiding as far as possible such operations. Some global commutative strategies do not ensure that an identical result reaches all processors. It must be remembered that a floating point operation has finite precision and so floating point arithmetic commutative operations are not truly commutative due to the effects of rounding errors. So for example a `GSUM` operation based on a ring of processors that accumulates partial summations by passing the partial results around the ring of processors will complete in $(P-1)t_l$ but the values left by `GSUM` on each processor will have different rounding errors. This can cause severe problems to many algorithms. If, for instance, the result of `GSUM` is tested to determine convergence some processors may test true and others false and the code will consequently fail. Execution of a global summation in parallel must produce a different result to the serial summation but both results are valid. It is only required that a global commutative produces an identical result across all processors, not an identical result to the serial commutative operation.

The `SCATTER` routine is used to distribute a variable across the processors, again in accordance with the given `SPATIAL_REFERENCE`. Similarly `GATHER` is used to rebuild variables from components on each processor. `SCATTER GATHER` operations are costly of both time and memory, requiring a number of messages proportional to P and globally dimensioned data space. The negative impact of these operations is however not particularly significant as they are only required for i/o operations.

4.3 Matrix Decomposition

Computational mechanics codes invariably require the solution to a number of systems of equations of the form $\mathbf{Ax} = \mathbf{b}$ which represent the discretisation of the equations governing the physical processes. For an element based finite difference method the form of the matrix equation for a regular 4×4 mesh is represented in Figure 4.4

Splitting this 4 *times*4 mesh into two renumbered sub-domains is illustrated in Figure 4.5. Here the parallel system matrices are no longer square as the rows of the matrices

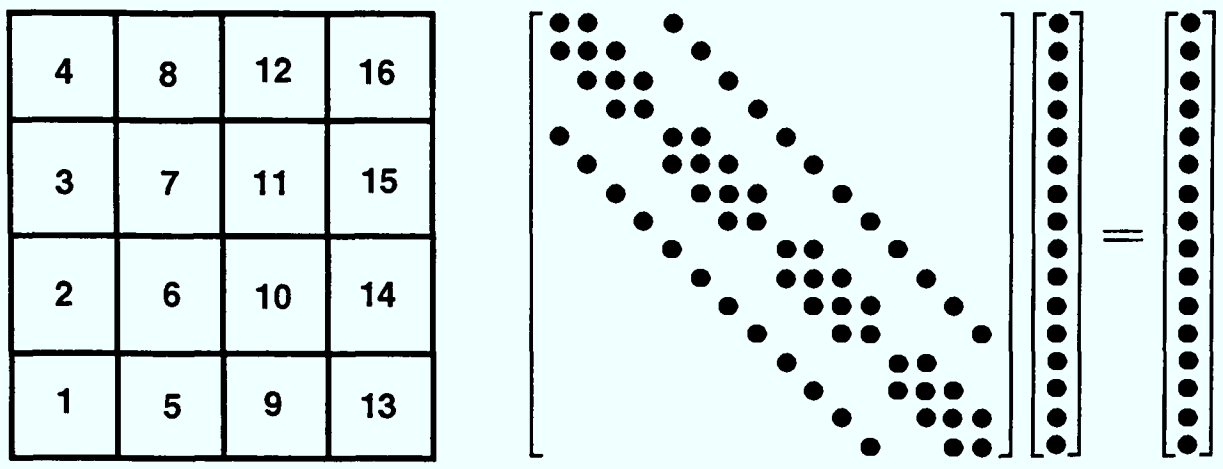

Figure 4.4: Matrix form for a five point element stencil over a 4×4 regular mesh.

that correspond to the inter processor boundary now contain coefficients that address elements that lie in the overlap region beyond the core length of the **x** vector. Note that the number of rows in the system matrix and the length of the **b** vector correspond with the number of core elements. The matrix and **b** vector are not required for the overlap. Note also that the 64 matrix coefficients are divided equally between each sub-domain. No additional calculation is required. The matrix and vector are constructed by each processor as if the processor was operating in isolation only upon its sub-domain core. The mesh topology for each sub-domain causes the generation of extra coefficients that correspond to the overlaps. The solution of the two sets of equations, one for each sub-domain achieves consistency through the interchange at each iteration of the solver of the coefficients of **x** that lie in the overlaps. The values for the overlap regions are exchanged as shown by the arrows in Figure 4.5.

These small, structured examples are easy to follow but do not clearly illustrate the effects of a decomposed system matrix for an unstructured mesh. Figure 4.6 illustrates a simple two dimensional unstructured mesh of 42 triangular elements.

Figure 4.7 shows the same mesh partitioned into three sub-domains which have been extended with a layer of overlap elements. The sub-domains are shown as being renumbered following the ordering of the original mesh. This is simply an aid to seeing how the

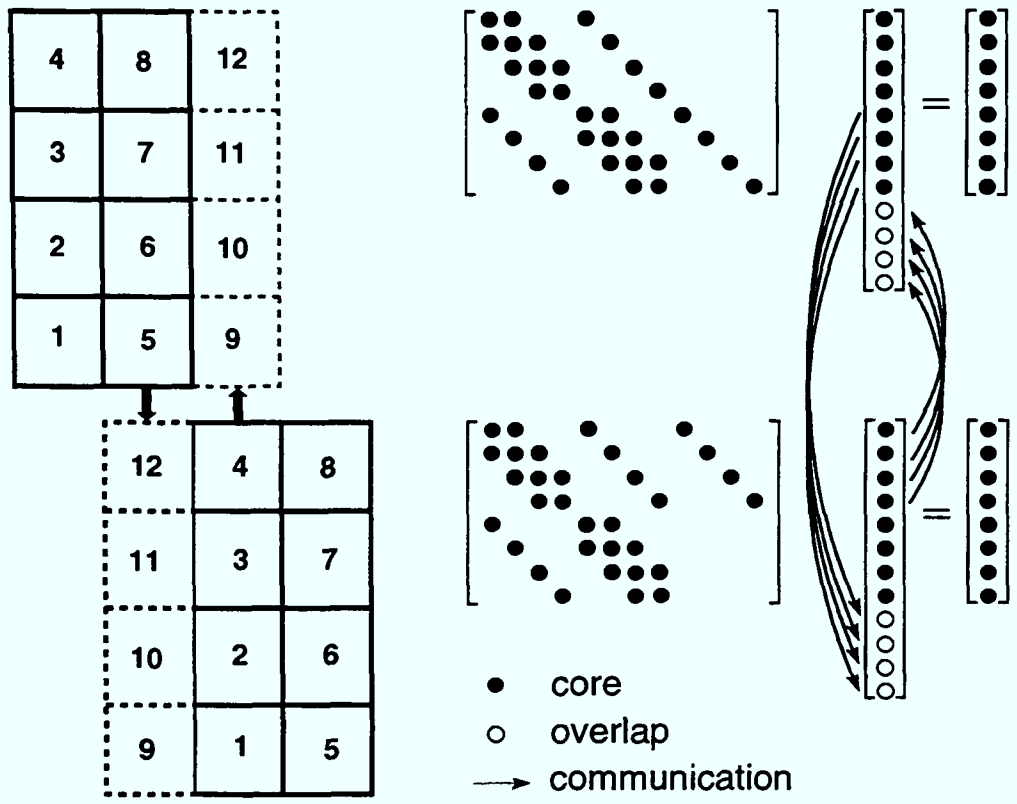

Figure 4.5: 4×4 mesh operated on as 2 sub-domains showing the transfer of data into the overlaps on each renumbered sub-domain.

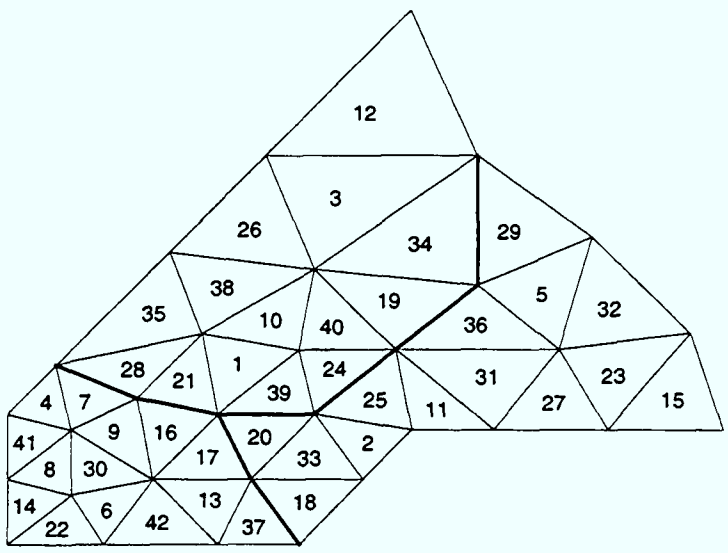

Figure 4.6: Mesh of 42 triangular elements.

system matrices for the decomposed problem have been constructed. The decomposed matrix would be the same had the sub-domains not been renumbered but the same element order followed. Even so this simple example is difficult for the eye to follow. Changing the order of the elements within the sub-domains would yield a different but nevertheless equivalent set of sub-domain matrix equations.

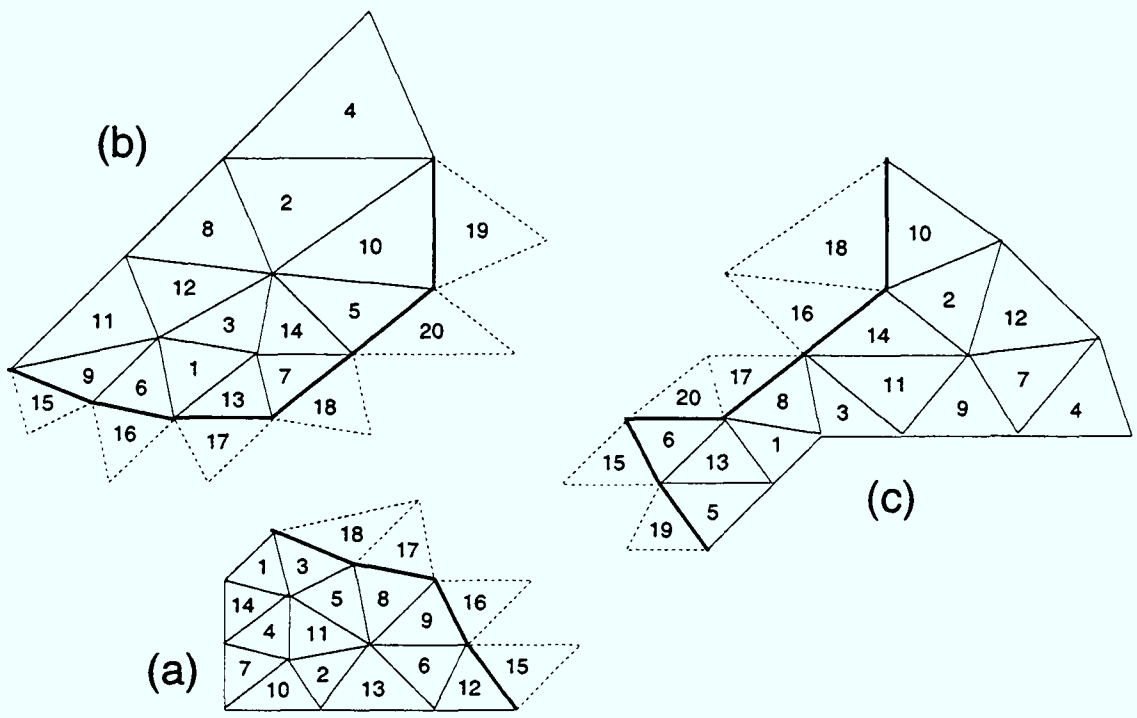

Figure 4.7: Mesh of 42 triangular elements partitioned into three renumbered sub-domains.

The original matrix is shown in Figure 4.8 to be sparse and irregular with a diagonally symmetric number of entries, as would be expected of an unstructured mesh problem. The matrix equations corresponding to the partitioned mesh are shown in Figure 4.9. This figure illustrates the complex pattern of dependence (communication) between the sub-domains (processors).

4.4 Iterative Methods

The length of $\mathbf{x}$ required by practical CM problems is large, of the order 1000 to 10,000,000. Iterative methods have been shown to provide the most effective and the

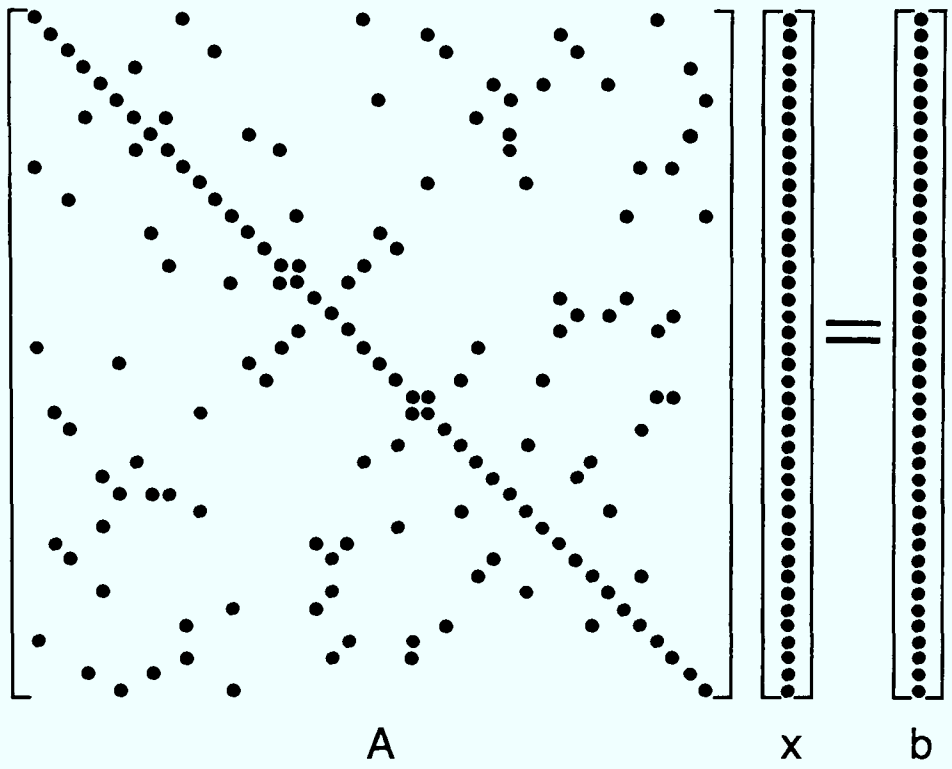

Figure 4.8: Matrix for the 42 triangle mesh.

most popular schemes. Direct methods tend to be more demanding of memory and less efficient when dealing with large problems. Three iterative methods are used by UIFS; Jacobi, Gauss Seidel SOR and the diagonally preconditioned conjugate gradient method.

4.4.1 Jacobi Method

The Jacobi method attempts to find a solution to $\mathbf{Ax} = \mathbf{b}$ by generating each $x_i^{(k+1)}$ from components of $\mathbf{x}^{(k)}$ for $k \geq 0$ according to Equation 4.30 until convergence is reached.

$$x_i^{(k+1)} = \frac{\sum_{j=1, j\neq i}^{n}(-a_{ij}x_j^{(k)}) + b_i}{a_{ii}} \qquad (4.30)$$

$$i = 1, 2, \ldots, n$$

This algorithm is entirely independent of the order in which the components $\mathbf{x}$ are evaluated, the values for $x_i^{(k+1)}$ are dependent only upon the values for the previous iteration $x_i^{(k)}$. In parallel each processor calculates a new vector $\mathbf{x}^{(k+1)}$ for its core components using the values for $\mathbf{x}^{(k)}$ in the core and overlap. The values for $\mathbf{x}^{(k+1)}$ are then copied into the overlap regions from the processors on which the components

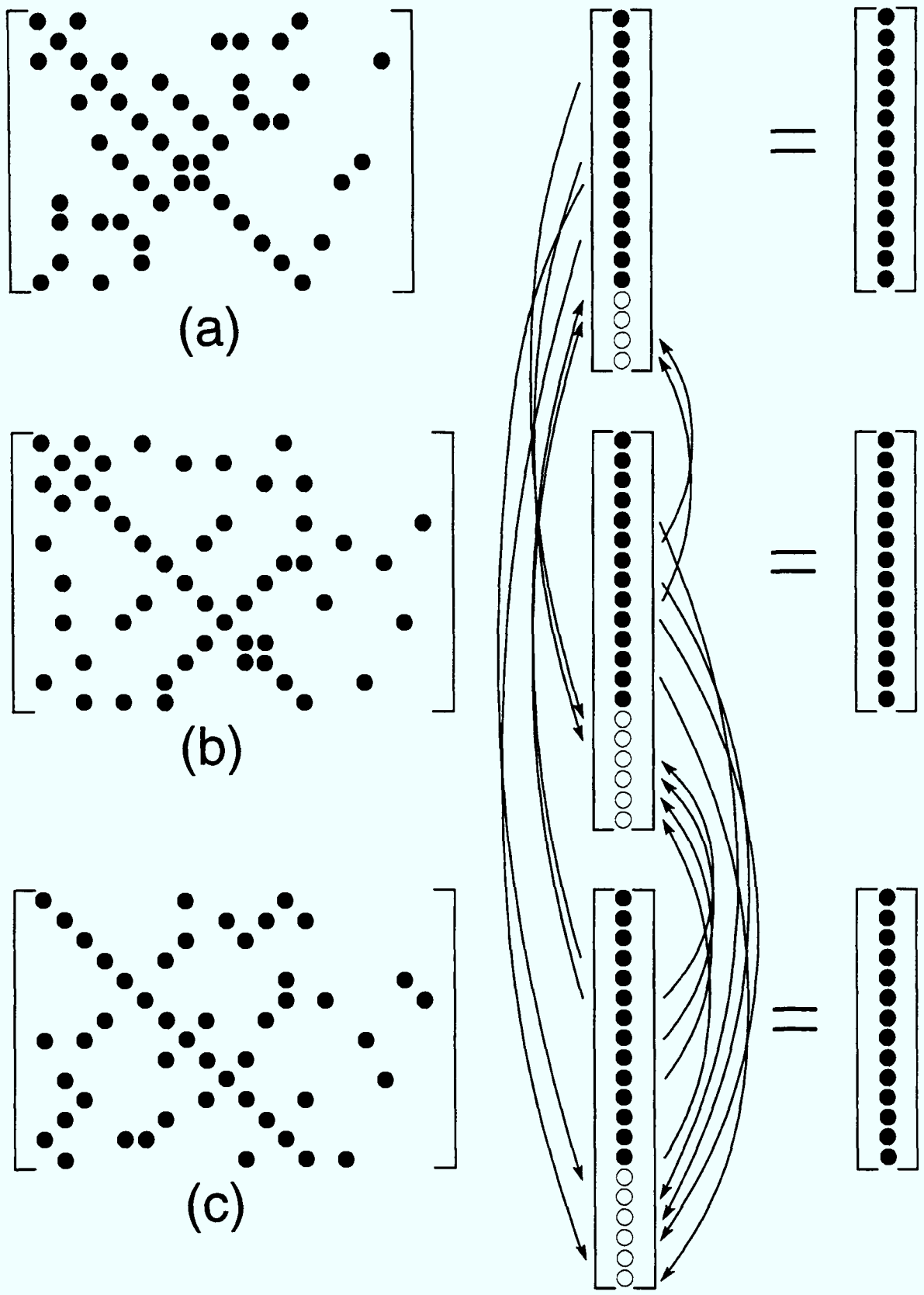

Figure 4.9: Matrices for the 42 triangle mesh partitioned into three sub-domains.

have been calculated. This process is carried out at each iteration to ensure consistency with the original serial algorithm. Using this parallel Jacobi solver the solution variables remain identical to those of the serial code at each iteration of the solution procedure. In parallel the system matrix is no longer a $n \times n$ sparse matrix as illustrated in Figures 4.4 and 4.8 but is partitioned and re-ordered (renumbered) as shown in Figures 4.5 and 4.9 to be distributed over P processors as a set of $n_p \times m_p$ sparse matrices where n_p is the number of elements (coefficients) in sub-domain p and m_p is the number of elements (coefficients) including the overlaps in sub-domain p. The parallel Jacobi method for P processors can therefore be expressed as Equation 4.31.

$$x_i^{(k+1)} = \frac{\sum_{j=1, j\neq i}^{m_p} (-a_{ij} x_j^{(k)}) + b_i}{a_{ii}} \qquad (4.31)$$

$$i = 1, 2, \ldots, n_p$$

$$p = 1, 2, \ldots, P$$

Equation 4.31 may give the impression that the parallel solver now has to loop over k, n_p and m_p, which would not be a particularly efficient parallel method. But the system matrix is sparse and the code for the solver only loops over non-zero coefficients in each row. The number of non-zero coefficients for each row (mesh entity) of the parallel system matrix is identical to that of the serial system matrix Consequently the parallel overhead caused by the non-square local matrices is zero.

Convergence is tested using the norm (l_1, l_2 or l_∞) of the difference between $\mathbf{x}^{(k+1)}$ and $\mathbf{x}^{(k)}$. The actual value of the l_1 and l_2 norms depend upon a global summation and so rounding errors could in theory cause the parallel version of the algorithm to terminate one iteration before or after convergence of the serial algorithm. This effect is however rarely observed in practice.

The parallel Jacobi algorithm is given in Appendix C.1.

4.4.2 Gauss-Seidel SOR

The Gauss-Seidel Successive Over Relaxation (GS-SOR) method was developed as an improvement of the Jacobi solver that typically exhibits faster convergence. The Gauss-Seidel method differs from the Jacobi method only in that the most current values of the variable $\mathbf{x}$ are used in each iteration. This can be thought of as overwriting $\mathbf{x}$ at each iteration which has the advantage of reduced storage requirement. Successive Over Relaxation is a scalar magnification factor α applied at each iteration in a attempt to accelerate convergence. GS-SOR may be expressed as Equation 4.32.

$$x_i^{(k+1)} = \alpha \left(\frac{\sum_{j=1, j\neq i}^{n} (-a_{ij} x_j^{(k+1)}) + b_i}{a_{ii}} \right) + (1-\alpha) x_i^{(k)} \qquad (4.32)$$

$$i = 1, 2, \ldots, n$$

An over relaxation coefficient may be similarly applied to the Jacobi method, however it is sometimes chosen to under relax a solver ($\alpha < 1.0$) in order to improve the stability of the algorithm. The optimal relaxation coefficient may be calculated using sophisticated eigenvalue analysis. Such analysis is however rarely performed, empirical values for α are generally adopted. Because the most current values are used for evaluation of each coefficient the algorithm is dependent upon the order of evaluation. This order dependency is sometimes used in structured problems as a means of accelerating convergence for some problems by 'upwinding' the solvers with the pressure gradient. When parallelising a Gauss-Seidel solver for a structured mesh, pipeline techniques may be used to ensure consistency of the parallel algorithm [JC91]. The ordered sweep of a solver across the domain, which makes such techniques possible is however not appropriate when considering an unstructured mesh. Parallel communication costs make it inefficient to *identically* parallelise a Gauss-Seidel iterative solver for an unstructured mesh as the parallelism is restricted and many small, frequent communications will be required. The order of evaluation of the coefficients must be modified if an effective parallel scheme is to be found. Mesh ordering may simply be a side effect of mesh generation or an attempt at cache optimisation but has no intended effect on the numerical scheme. Alteration of

the order of coefficient evaluation is therefore of little consequence (bandwidth minimisation techniques may be applied to the decomposed matrices). The simplest and most obvious solution is to implement an overlap update scheme exactly as described for the Jacobi algorithm. The resulting parallel algorithm becomes a near Gauss-Seidel hybrid of Gauss-Seidel and Jacobi in that the components of $\mathbf{x}^{(k+1)}$ that are addressed in the overlaps are actually $\mathbf{x}^{(k)}$. This may not be so great a disturbance to the algorithm as it first appears. Equation 4.32 is not particularly accurate as the coefficients of $x_j^{(k+1)}$ in are actually $x_j^{(k)}$ for $i < j$. The GS-SOR algorithm may be more correctly expressed as:

$$x_i^{(k+1)} = \alpha \left(\frac{\sum_{j=1}^{i-1}(-a_{ij}x_j^{(k+1)}) + \sum_{j=i+1}^{n}(-a_{ij}x_j^{(k)}) + b_i}{a_{ii}} \right) + (1-\alpha)x_i^{(k)} \qquad (4.33)$$

$$i = 1, 2, \ldots, n$$

In a similar manner to Equation 4.31 the parallel Gauss-Seidel SOR equation implemented over P processors can be expressed as:

$$x_i^{(k+1)} = \alpha \left(\frac{\sum_{j=1}^{i-1}(-a_{ij}x_j^{(k+1)}) + \sum_{j=i+1}^{m_p}(-a_{ij}x_j^{(k)}) + b_i}{a_{ii}} \right) + (1-\alpha)x_i^{(k)} \qquad (4.34)$$

$$i = 1, 2, \ldots, n_p$$

$$p = 1, 2, \ldots, P$$

Results given in this thesis show that variations in the values of serial and parallel variables and differences in the number of iterations required to converge are both insignificant. In practical terms the variations between the serial and parallel results are significantly less than the variations caused by running the serial code on different processors (Sparc, i860, MIPS, etc.). Even with processors using IEEE arithmetic differences in rounding modes lead to variations in results.

The parallel Gauss-Seidel SOR algorithm is given in Appendix C.2.

4.4.3 Conjugate Gradient

The Conjugate Gradient (CG) method has become an established nonstationary iterative method for symmetric positive definite systems due to its rapid convergence rate and computational efficiency, $O(m)$ where m is the number of non-zero components of $\mathbf{A}$. The conjugate gradient solver is an extension of the method of steepest descent where search directions are constructed by conjugation of the residuals [She94, GL89, BBC+94]. Preconditioning is often applied to improve the condition number of the matrix $\mathbf{A}$. For a positive definite matrix preconditioner $\mathbf{M}$

$$\mathbf{Ax} = \mathbf{b} \quad \equiv \quad \mathbf{M}^{-1}\mathbf{Ax} = \mathbf{M}^{-1}\mathbf{b} \tag{4.35}$$

If the eigenvalues of $\mathbf{M}^{-1}\mathbf{A}$ are clustered better than the eigenvalues of $\mathbf{A}$ then the preconditioned problem may be iteratively solved faster than the original problem. A clear description of preconditioning is given by Shewchuck in [She94]. The preconditioned conjugate gradient algorithm consists of iterating the following stages until $\frac{\rho^{(k+1)}}{\rho^{(1)}}$ reaches the required precision.

$$\mathbf{r}^{(1)} = \mathbf{b} - \mathbf{Ax}^{(1)} \tag{4.36}$$

$$\mathbf{p}^{(1)} = \mathbf{M}^{-1}\mathbf{r}^{(1)} \tag{4.37}$$

$$\rho^{(1)} = \mathbf{r}^{(1)T}\mathbf{p}^{(1)} \tag{4.38}$$

$$\mathbf{u}^{(k)} = \mathbf{Ap}^{(k)} \tag{4.39}$$

$$\alpha^{(k)} = \frac{\rho^{(k)}}{\mathbf{p}^{(k)T}\mathbf{u}^{(k)}} \tag{4.40}$$

$$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \alpha^{(k)}\mathbf{p}^{(k)} \tag{4.41}$$

$$\mathbf{r}^{(k+1)} = \mathbf{r}^{(k)} - \alpha^{(k)}\mathbf{u}^{(k)} \tag{4.42}$$

$$\mathbf{z}^{(k+1)} = \mathbf{M}^{-1}\mathbf{r}^{(k+1)} \tag{4.43}$$

$$\rho^{(k+1)} = \mathbf{r}^{(k+1)T}\mathbf{z}^{(k+1)} \tag{4.44}$$

$$\beta^{(k+1)} = \frac{\rho^{(k+1)}}{\rho^{(k)}} \tag{4.45}$$

$$\mathbf{p}^{(k+1)} = \mathbf{z}^{(k+1)} + \beta^{(k+1)}\mathbf{p}^{(k)} \tag{4.46}$$

This method involves three basic computational processes; matrix-vector product, vector inner product and AXPY ($a\mathbf{x}$ plus $\mathbf{y}$).

Remembering that each distributed $\mathbf{A}$ matrix is no longer square (Figure 4.9) as it now addresses coefficients in the overlaps. So an overlap exchange communication is required to obtain the values of p in the overlaps before evaluating the matrix vector product $\mathbf{Ap}$ in equation 4.39.

The inner products in equations 4.40 and 4.44 are calculated in parallel as a sum of local partial inner products. Equation 4.44 for example is evaluated as:

$$\rho = \sum_{p=1}^{p=P} \sum_{j=1}^{j=m_p} r_p(j) z_p(j) \tag{4.47}$$

This requires a global summation and hence synchronisation across all processors.

The AXPY in equations 4.41, 4.42 and 4.46 is an ideally parallel process requiring no inter processor communication.

The simplest preconditioner is the diagonal or Jacobi preconditioner which is the diagonal of the $\mathbf{A}$ matrix that has the effect of scaling the quadratic form along the co-ordinate axes. Whilst not the most effective preconditioner this is easy to implement and effective for most reasonably well conditioned CM matrices. The actual CG method used in UIFS uses a diagonal prescaling modification [LL88] which involves transformation of $\mathbf{Ax} = \mathbf{b}$ into $\check{\mathbf{A}}\check{\mathbf{x}} = \check{\mathbf{b}}$ where the components of $\check{\mathbf{A}}$, $\check{\mathbf{x}}$ and $\check{\mathbf{b}}$ are:

$$\check{a_{ij}} = \frac{aij}{\sqrt{a_{ii}a_{jj}}} \tag{4.48}$$

$$\check{x_i} = x_i\sqrt{a_{ii}} \tag{4.49}$$

$$\check{b_i} = \frac{b_i}{\sqrt{a_{ii}}} \tag{4.50}$$

This results in the diagonal of $\check{\mathbf{A}}$ being the identity matrix $\mathbf{I}$ and so for a Jacobi preconditioner $\mathbf{M} = \mathbf{I}$. This has the computational advantages of removing equation 4.43 and simplifying calculation of the matrix-vector product in equation 4.39. After convergence $\check{\mathbf{x}}$ is rescaled to give $\mathbf{x}$.

$$x_i = \frac{\check{x_i}}{\sqrt{a_{ii}}} \tag{4.51}$$

The Diagonally Preconditioned Conjugate Gradient (DPCG) algorithm, along with most other preconditioning schemes is explicit in that it uses only old variable values within each iteration. It may therefore be expected to give identical results from both serial and parallel versions. However rounding errors occur in the global summation involved in the inner products and these errors are different for serial and parallel implementations. As the solutions are highly sensitive to α and β these small variations lead to differences between the serial and parallel solution. In this case both serial and parallel solutions are equally valid solutions to the original problem. Much of the literature discusses the efficient implementation of global accumulation [dC95] without mention of this effect.

The parallel diagonally preconditioned conjugate gradient algorithm is given in Appendix C.3.

4.4.4 Summary

Implementation of geometric domain decomposition as presented in Chapter 3 within UIFS was entirely straightforward. The *entire* UIFS code has been parallelised with only minimal changes to the code and the algorithm being required. Many of the subroutines required no changes whatsoever as was anticipated in Section 3.3.3 The majority of programming effort was required for the implementation of the initial decomposition of the problem. The three iterative methods discussed, Jacobi, Gauss SOR and DPCG provide algorithms with a high degree of easily utilised parallelism. The Jacobi method is one of the simplest algorithms to parallelise. It requires only one exchange of overlap data per iteration and one global operation to determine convergence. The DPCG algorithm appears at first sight to be similarly straightforward. However the global summations involved in the algorithm affect the numeric result. It is worth remembering that a simple arithmetic process such as summing n real numbers is affected by rounding errors. The result given by summing from 1 to n is likely to be different from the result of summing from n to 1. In parallel, with two processors the summation would be executed as something like $\sum_1^{\frac{n}{2}} + \sum_{\frac{n}{2}+1}^{n}$, which would again give a different result. In practice

the coefficients that constitute the system matrix are also subject numerical differences arising from rounding errors which can mask the rounding effects from the solver. If the original serial algorithm is stable then these effects have no actual significance on the results. If rounding effects lead to divergence of the parallel results from the serial then suspicion must fall on the validity of the serial case.

The Gauss SOR scheme is however subject to algorithmic modification. There are schemes that can allow this algorithm to be faithfully reproduced in parallel but such schemes involve frequent small communications and/or pipelining techniques with the consequent high cost of communication startup latency along with pipeline startup and shutdown latency. Given that parallel machines are far from perfect the more pragmatic parallel scheme described in Section 4.4.2 has been successfully adopted.

Chapter 5

Performance of the Parallel Code

The performance yardstick for a parallel code is often by comparison of the run-time for one processor t_1 against the run-time for many processors t_p. This gives rise to a number of interesting problems. For example it may not be possible to run a large problem on only one processor, or indeed small numbers of processors, if it does not fit into the available memory. It is possible that modification of the algorithms may be required to achieve a parallel solution. In which case the the run-time for the best serial code on one processor should be compared with the parallel run-time [RVD93]. Such results are highly machine dependent. The calculation to communication ratio of a machine has a profound effect on the parallel performance of a particular code. Early developments in this research were conducted on T800 transputer based equipment which returned very good parallel efficiency. Rather than reflecting a good parallel solution these results reflected the rather poor calculation performance of the T800 in comparison to its good communication capability. Results are highly problem dependent. Problem size can determine whether latency or bandwidth forms the bound on performance. Some workers prefer a more absolute frame of reference such as comparison of the run-time of a problem on a parallel machine with a well known serial machine, often a Cray Y-MP. This reduces to a measure of the achieved Mflop rate. Additionally scalability tries to provide some measure to describe how far the parallelism of a code and/or platform may be exploited, i.e. does the performance scale with the number of processors? Invariably the parallel

performance is a function of the nature of the machine, the original code and the quality of the parallelisation.

5.1 Measuring Performance

Strictly speaking the run time of the original serial code should be used as a measure of the run time on one processor. This is however not always the most practical scheme. It is often the case that in scrutinising a code for parallelisation there arise instances where optimisations of the serial code may be made, and must be made to achieve honest comparisons. One common occurrence in CM codes is the printing of end of sweep residuals, principally as a means of imparting confidence to the code user. Interrupting an operating system to print can carry a significant overhead and so silencing a code gives a reduced run-time. This effect is of greater importance in parallel where for many systems the operating system interrupt can carry a significant overhead. We are left with a dilemma as to what we consider to be the run time on one processor and what is the run time on many processors. Many CM codes incorporate a timer to report the elapsed CPU time for a run. It has become normal practice for such timers to start after reading the problem specification from file and stop before writing results to file. This is reasonable as file access times can be dependent on other traffic on the systems. Timing only the CPU activity gives an optimistic view of parallel performance as parallel i/o hardware is rare and so i/o activity seldom scales. The order of CM codes tends to be somewhere between linear $O(N)$ and quadratic $O(N^2)$ so measuring only CPU time is not unreasonable as CPU time forms the asymptotic bound on run-time for large problems.

This situation can become difficult when faced with parallelisation of codes that perform unnecessary calculations. That is computation that has no effect whatsoever on the results. There is a choice between identically parallelising the unnecessary calculation or modifying the serial code to remove the redundant code. It is possible that the redundant code can involve dependencies across the mesh that are not required by the

rest of the code. It is often the case that some 'fixing' of the serial code is required in the parallelisation process.

The results presented in this Chapter use the CPU time of the parallel code on one processor for t_1, which is in this case less than the run time of the original code. The overhead of the parallel version on a single processor is only the cost of the call to the communication routines in which no communication occurs. This has proved to have an insignificant impact on the run time in numerous parallelised codes.

5.1.1 Speed-up

Parallel speed-up S_P is the ratio of the run-time on one processor t_1 to the run-time on P processors t_P.

$$S_P = \frac{t_1}{t_P} \tag{5.1}$$

If the parallelisation is 100% efficient then $S_P = P$ but this is rarely the case for real CM problems. There is always some fraction of the code f_s $(0 < f_s < 1)$ that is inherently serial. This limitation on the maximum possible speed-up S_P^{max} is summarised as Amdahl's law in Equation 5.2 [Amd67].

$$S_P^{max} = \frac{t_1}{\frac{(1-f_s)t_1}{P} + f_s t_1} \tag{5.2}$$

The asymptotic limit of Amdahl's law as $P \to \infty$ gives:

$$S_P^{max} = \frac{1}{f_s} \tag{5.3}$$

This clearly places a finite limit on the maximum achievable speed-up from a parallel code. Amdahl's law has been cited as a strong reason to doubt the usefulness of massively parallel systems. For a fixed problem size f_s is constant and so scalability is restricted. Scalability can only be possible if f_s reduces with an increasing problem size. In practice f_s for a CM code is often extremely small even with modest problem sizes. CM codes tend to be somewhere between $O(N)$ and $O(N^2)$ whereas f_s somewhere between constant and $O(N)$. Consequently f_s tends towards insignificance as the problem size increases and so scalability becomes possible. The communication cost and the

idle time invariably suffered in a parallel code also deteriate the performance further, however other factors not included in Amdahl's law such as better cache usage for each sub-domain in comparison with the global problem can have a beneficial effect.

5.1.2 Parallel Efficiency

Parallel efficiency is sometimes used as the performance measure for a parallel code. Parallel efficiency E_P is simply the ratio of the parallel speed-up S_P to the number of processors P.

$$E_P = \frac{S_P}{P} \times 100\% \tag{5.4}$$

As Section 5.1.1 has shown parallel efficiency cannot exceed 100%, or can it? There are two instances in which parallel efficiency may become 'superlinear' and exceed 100%. One possibility is to break some data dependency in the parallel code that is not actually required. The implication here is that the serial code is open to some form of optimisation. Having applied the optimisation to the serial code a superlinear parallel efficiency should no longer be achievable. The other cause of superlinear performance is cache usage. Decomposing a large problem, that does not fit well into cache, into a number of small problems, may allow the decomposed problems to fit into cache. Cache success is an important factor in CPU performance, especially for the extremely high clock rate (>100MHz) new generation of processors that are able to process data far faster than conventional DRAM memory may be accessed.

5.1.3 Scalability

There are two aspects to scalability; scalability of computation and scalability of memory.

Scalability of Computation

Computation is said to be scalable if the gradient of the graph of speed-up against number of processors is positive. That is if more processors are used then the run time will reduce. In practical terms the returned processing power is not profitable once the gradient of the curve has reduced to around 0.5. Given that practical problems must have

an inherently serial portion of code then a given problem has a finite limit on scalability dictated by Amdahls law. It is often chosen to demonstrate the scalability of a machine using a constant (usually large) problem size *per processor* in an attempt at minimising the appearance of Amdahls law. For a fixed problem, as the number of processors increases, the compute time on each processor decreases while the communication time remains constant or increases slightly [Joh92]. It is predominantly this change in the ratio of calculation to communication that leads to the drop in speed-up as the number of processors increases.

Scalability of Memory

Memory is said to be scalable if the problem size can be increased in proportion to the number of processors. That is (assuming a constant amount of memory per processor) if the number of processors is doubled can a problem twice as big be accommodated on the machine. Scalable memory implies that there are no globally sized data items and no significant arrays that have the number of processors as an index. In practice it can be simplifying to have some global sized structures. With care the restriction on memory scalability can be an acceptable level. For example the topology of a hexahedral element can be represented by an array of length number of elements and width eight. If the memory size on each PE is x words then the largest topology that can be held on one PE is $\frac{x}{8}$. A typical code may use around 100 variables each of length number of elements. So the largest problem that can be run per PE will be $\frac{x}{100}$. If this memory space is to be used, for example, to store the entire mesh topology for the purpose of partitioning and then re-cycled to hold the distributed problem data, this places a limit on scalability of $\frac{100}{8}$, or 12 processors, which is clearly not acceptable. If however only a single globally dimensioned vector is required and the code uses 200 variables then the limit on scalability is more like 200 which could well be considered acceptable. If the code uses more variables, or i/o processors are available with increased memory then this limit can easily become larger than any currently available machine. So with a little care it is possible to take advantage of a globally dimensioned data structure without

prejudicing scalability.

5.2 Irregular Shape Test Case

The irregular shape mesh of 3034 triangular elements partitioned by JOSTLE using three different mapping strategies is shown in Figure 3.4. This shape was automatically meshed [Law94] as 3034, 10027, 30064, 60005 and 119822 triangles. The JOSTLE code [Wal95] was used to partition each of the meshes using five different partitioning strategies:

i) Unmapped: Machine topology is ignored throughout the partitioning process.

ii) Postmapped: The unmapped partition is post-mapped to match the machine topology as a $p \times 2$ grid.

iii) Premapped: Initially mapped 2D partition optimised to reduce the number of cut edges.

iv) Mapped1D: Mapped to a 1D processor array.

v) Mapped2D: Mapped to a 2D processor array.

The effect of the partitioning strategy on the cut edge count is shown in Figures 5.1 – 5.5. Through all of the mesh sizes the lowest cut edge count is obtained using the unmapped (postmapped) partitioning strategy. The mapped1D and mapped2D partitions give the highest cut edge count with the mapped1D partition having approximately twice the cut edge count of the other partitions.

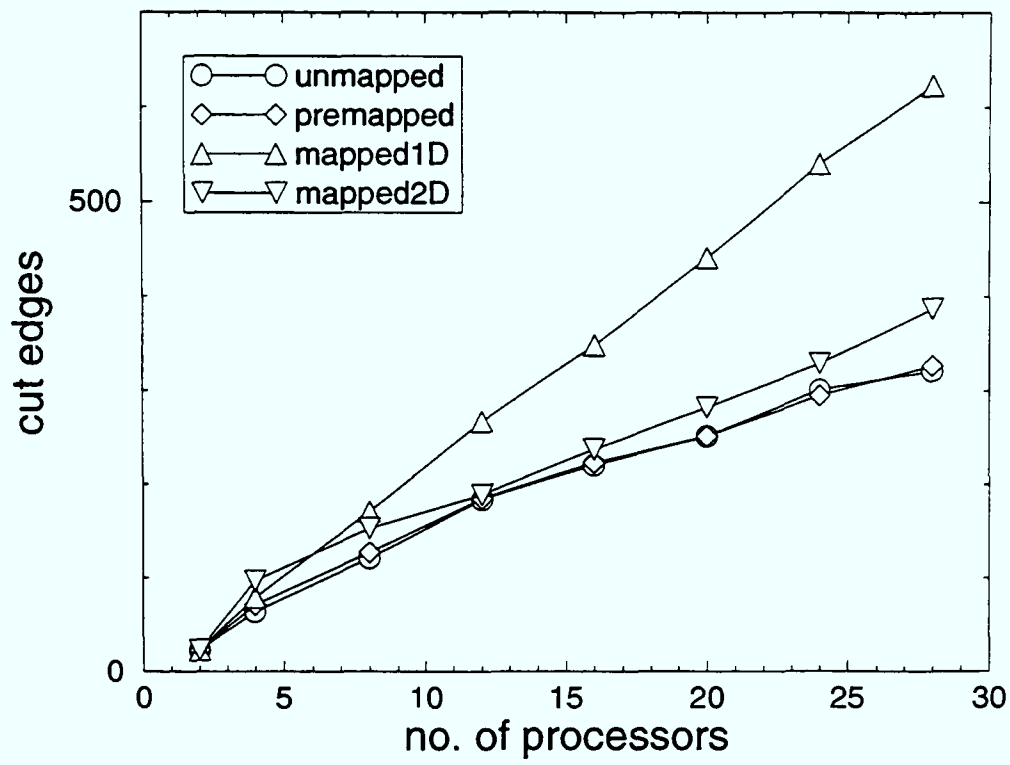

Figure 5.1: The number of cut edges against the number of partitions for a range of partition strategies on the 3,034 triangle irregular shape mesh.

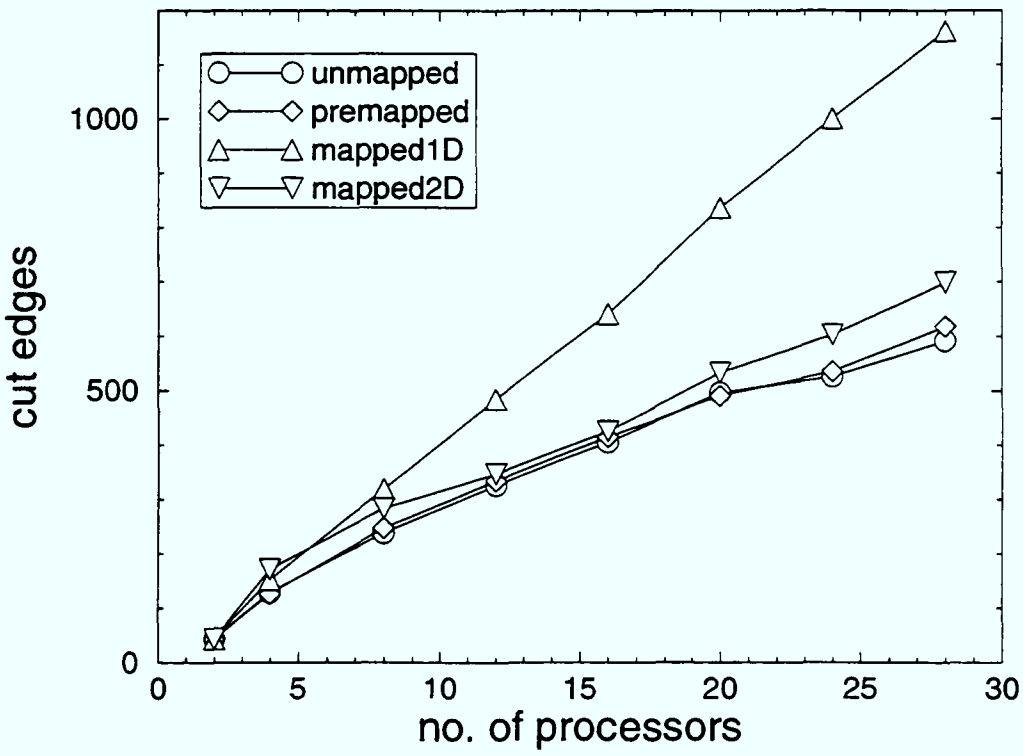

Figure 5.2: The number of cut edges against the number of partitions for a range of partition strategies on the 10,027 triangle irregular shape mesh.

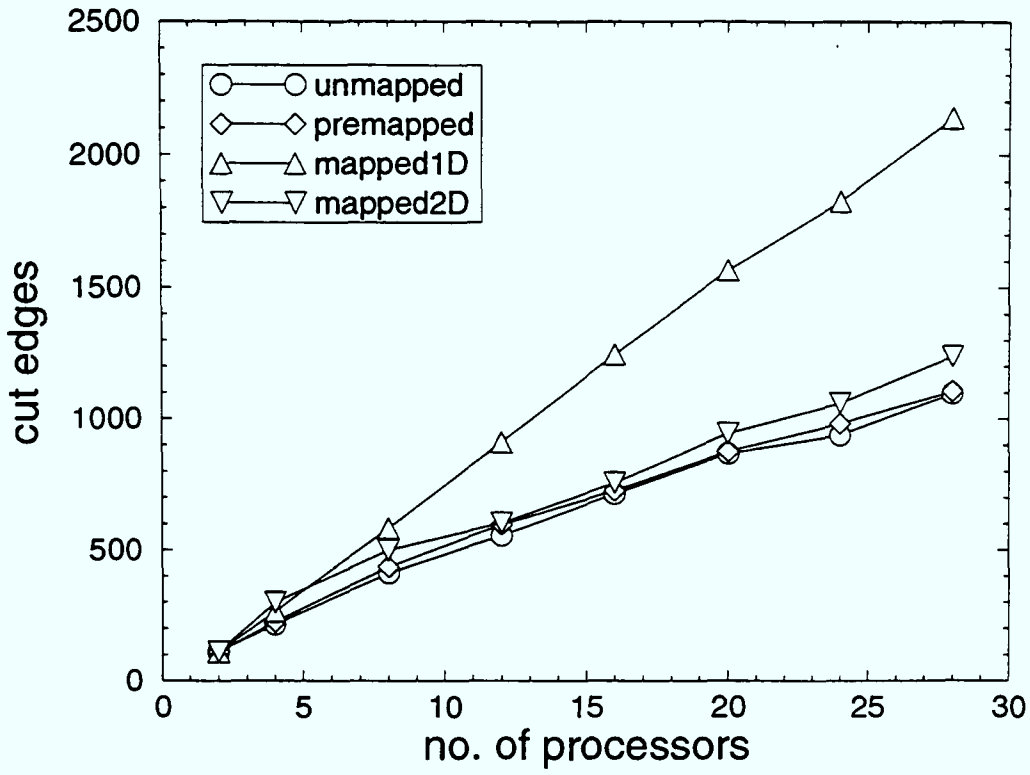

Figure 5.3: The number of cut edges against the number of partitions for a range of partition strategies on the 30,064 triangle irregular shape mesh.

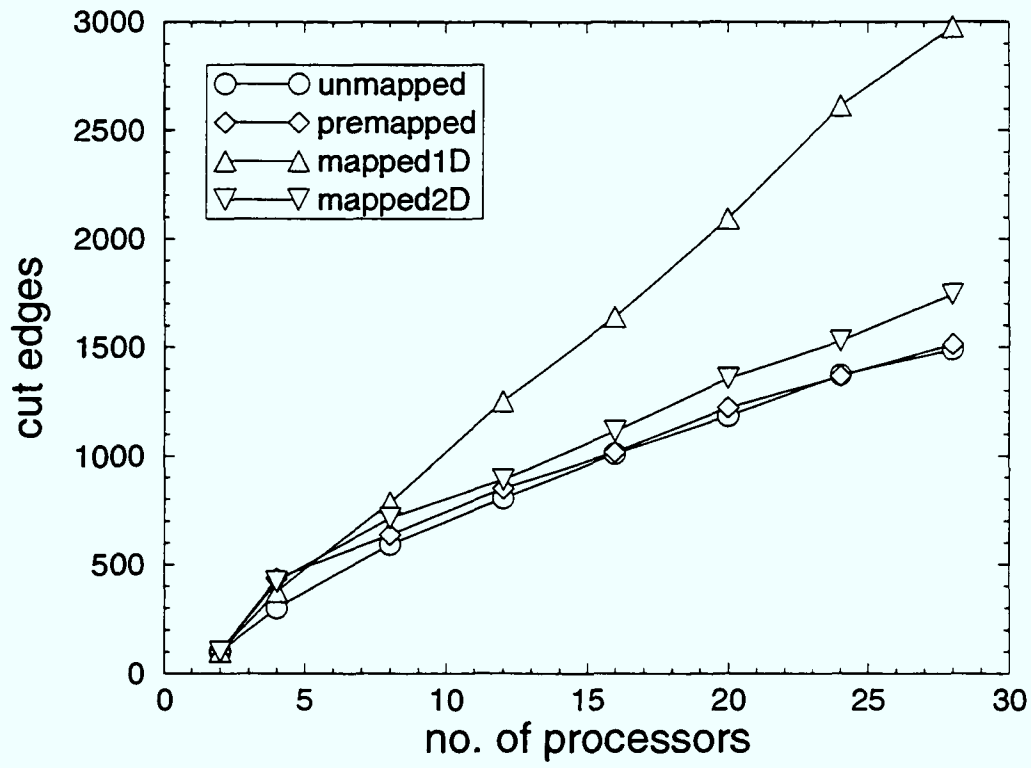

Figure 5.4: The number of cut edges against the number of partitions for a range of partition strategies on the 60,005 triangle irregular shape mesh.

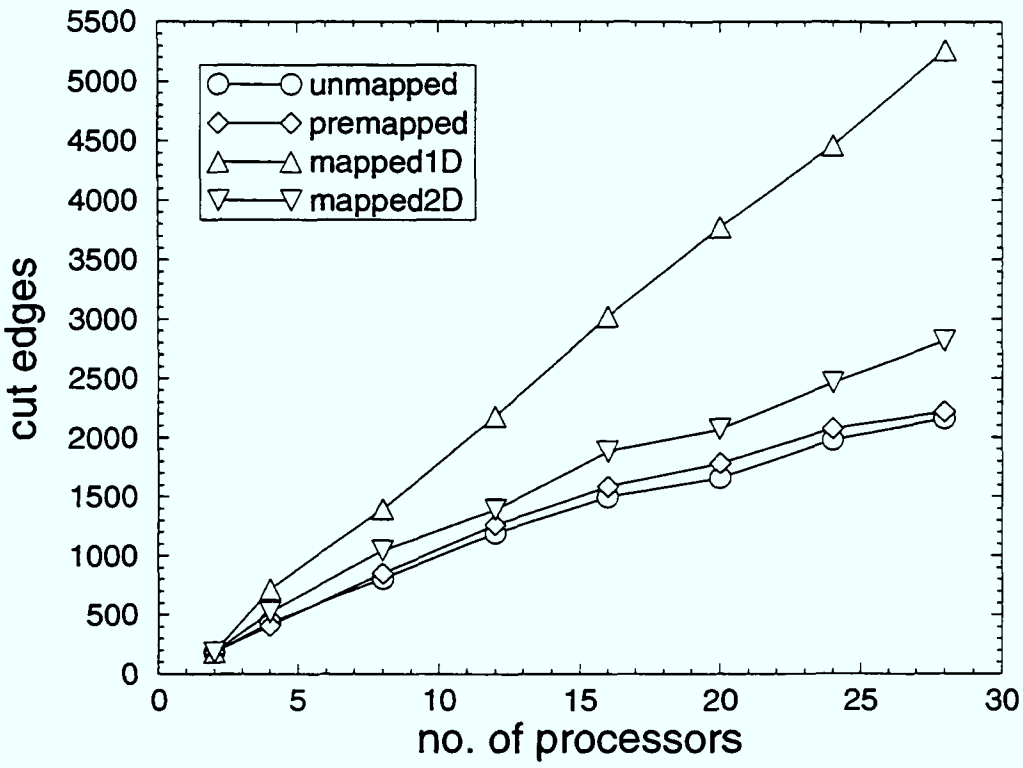

Figure 5.5: The number of cut edges against the number of partitions for a range of partition strategies on the 119,822 triangle irregular shape mesh.

temperature of -30 Centigrade. This load was applied for four two second time steps. Four time steps were used simply to provide a convenient run time for the purposes of measurement. An exaggerated mesh displacement is shown in Figure 5.7. Only the displacements are solved in this test case, stresses being calculated from the displacements. The diagonally preconditioned conjugate gradient method is used in the displacement solvers.

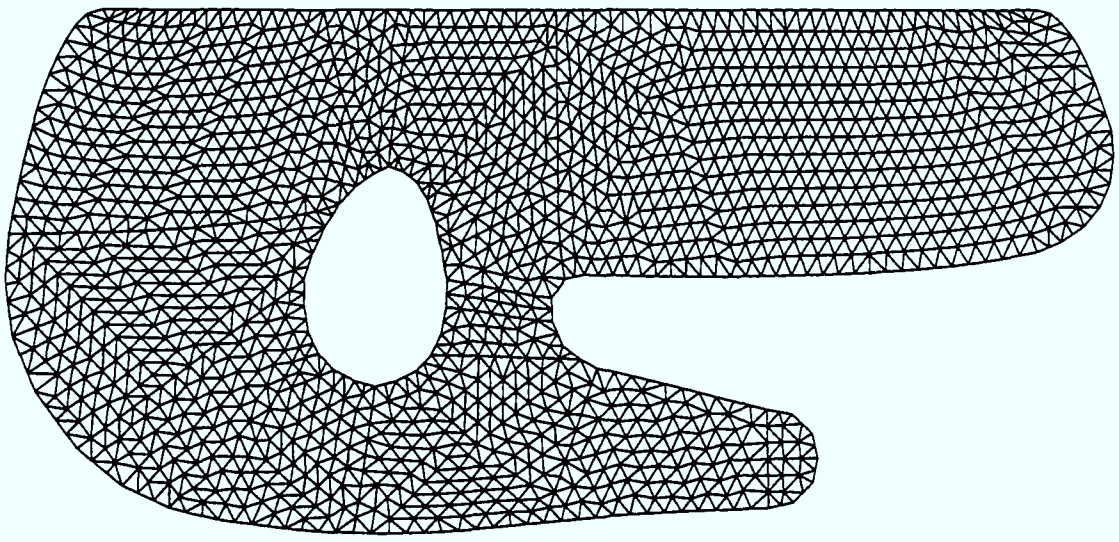

Figure 5.7: Mesh displacement for the solid mechanics test case.

5.2.3 Solidification Test Case

The solidification test case starts with liquid gallium close to solidification at 30 Centigrade. The boundary is held at 20 Centigrade with the exception of the top surface which is held at 0 Centigrade. The case is run until the gallium is largely solidified with small patches of recirculating liquid remaining. The residual stress contours, mesh displacement and flow vectors are illustrated in Figure 5.8. This case uses the larger stress overlaps for both the flow and the stress portions of the problem. At the start of the run there is negligible work for the stress solver as the majority of the domain is liquid. At the end of the run only a small portion of the problem remains liquid yet the majority of the compute time is still required in the flow solvers. All of the solvers are enabled

Parmacs, PVM and C Toolset style communication libraries are all available on the Paramid. These results have been obtained using the C Toolset library as this library gives better performance than the alternatives on this platform.

The 30,064 element test case is the largest of the test cases that can fit into the memory of one 32MByte processor node. The serial run time for the 60,005 element case was regressed from the two processor run time and for the 119,822 element test case the four processor run time was used. Clearly this affects the absolute accuracy of the graphs but does not change the nature of the graphs in providing a comparison between partitioning techniques.

The lowest number of cut edges and therefore the lowest amount of communication for each mesh size is given by the unmapped (postmapped) partition but this partition clearly does not give the best speed-up performance. The unmapped and postmapped partitions are actually the same partition, the postmapped partition having had an additional optimised mapping of partitions to processors applied to it. Where the two partitions give a similar speed-up this reflects an unintentionally fortuitous mapping of the unmapped partition to the processor topology. It is possible that the unmapped and postmapped partitions may by chance be identical, it is however highly unlikely that the unmapped partition would ever give a better speed-up than the postmapped partition, in such a case the processor allocation strategy would have failed. Of course any performance differences between the unmapped and postmapped partitions are unlikely to be significant for small numbers of processors.

The best overall speed-up performance in the graphs is given by the mapped partitions, despite the cut edge count being higher than the other partitions. This confirms the proposition that partitioning in accordance with the machine topology will result in improved performance.

Using a pipelined (mapped1D) partition leads to a significantly higher number of cut edges and consequently the message length is far greater, however fewer messages are required. A mapped1D partition requires only two messages and hence two latencies for each overlap update (one to each neighbour), which explains the perhaps unexpectedly

good speed-up results for the pipeline partition.

The mapped2D partition in Figure 3.4 shows the maximum node degree of the processor communication graph to be four. However the edges in this communication graph represent only element adjacency but the data dependency is actually more extensive than merely adjacency. Adding overlaps to the sub-domains therefore increases the maximum node degree of the processor communication graph to five as the overlaps reveal dependencies between sub-domains previously shown as unconnected. Consequently five messages are required for each overlap update. Given that the imbalance of elements between the sub-domains for all cases is less than 0.25%, and for the secondary grid point partition the imbalance is less than 0.75%, the effect of load imbalance for the test cases is insignificant (constant element shape with near constant mesh density).

It is therefore apparent from these results that the machine performance with this code is latency bound for the smaller test cases and bandwidth bound for the larger flow dominated test cases. Consider Figure 5.9, here the best speed-up is given with the mapped1D partition, this partition has the greatest amount of data to communicate but the lowest number of messages (latencies) per processor. Clearly latency is the bound on performance with this problem. For the larger fluid dynamic test case shown in Figure 5.13 the mapped2D partition gives the best speed-up. Here the large amount of data communication required for the mapped1D partition is eroding the advantage of fewer latencies allowing the mapped2D partition to outperform it. Clearly the inter-processor bandwidth is the bound on this problem. For the graphs between the small and large test case the transition from latency to bandwidth bound can be seen. Figure 5.26 is an encouraging result that demonstrates that scalability is achievable given a large enough problem size. The slow down exhibited with the small test cases is a direct consequence of the communication dominating the calculation, as the number of processors increases the time required for calculation falls but the time required for communication remains more or less constant.

Investigation shows that the relatively poor results for the solid mechanics test cases are primarily a consequence of the two global commutative operations required in every

iteration of the the CG solver as implemented in the serial code. Each global commutative operation incurs a number of communication start-up latency costs, a high latency cost leads to poor performance. This is clearly revealed by profiling the parallel code execution where the global commutative summations dominate the run time. The solidification test case uses the larger overlaps required for the stress code but this has only a slight effect on the speed up in comparison with the flow only results. This confirms that the predominant limiting factor for performance on the Transtech Paramid is the communication start up latency. Part of the solidification test case involves the CG solver but again this only marginally affects the results as the time required for the stress calculation is considerably less than the time required for the flow and heat calculation.

Start-up latency on the Transtech Paramid has been measured as 33μs with a peak bandwidth of 1.7MBytes per second. This bandwidth is not sustained with virtual channel routing and degrades to around 1.3 for near neighbour communication and can get as low as 0.9 for non local messages. This can deteriorate further to around 0.3MBytes per second if the communication channels are saturated as they will be for real problems with unmapped partitions. Similarly the startup latency degrades with increasing network traffic. While this bandwidth is low in comparison with other parallel machines [DD95] the latency appears reasonably good. Similar performance may therefore be expected from other parallel platforms for the test cases that run to a latency bound. The test cases that show that what is bandwidth limitated on the Paramid would be expected to run slightly faster on other platforms and become latency bound.

Partitioning onto a $p \times q$ processor array where $q > 2$ has yet to be tested, but is not expected to improve performance on the Paramid (or indeed other machines) with these test cases because of the latency bound . Whilst a $q = 2$ mapped partition is likely to incur five latencies, a $q > 2$ mapped partition will incur eight latencies, but will not significantly reduce the number of cut edges until P (and N) increases considerably.

5.3.1 Fluid dynamic test case

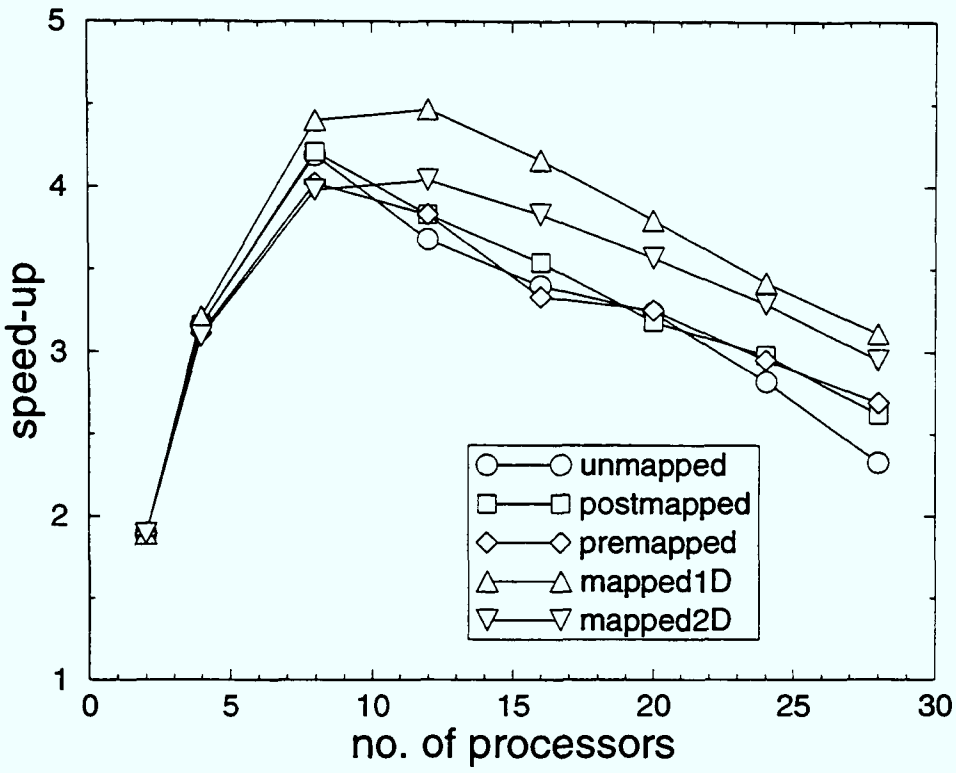

Figure 5.9: Speed-up for the fluid dynamic test case against the number of processors for a range of partition strategies using a 3,034 triangle mesh.

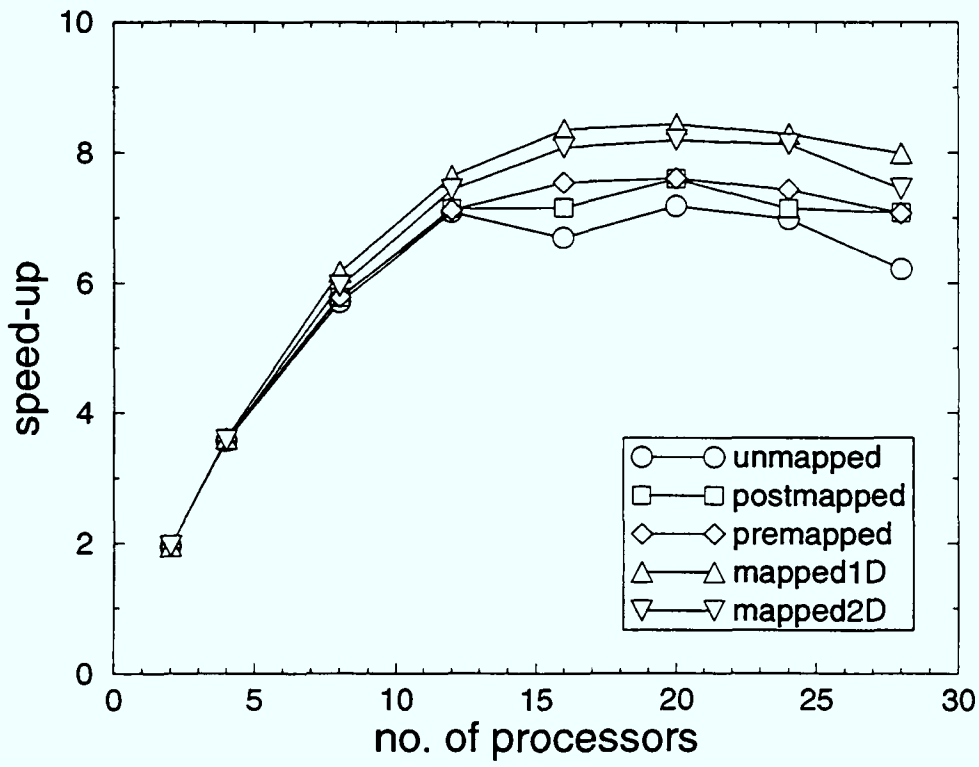

Figure 5.10: Speed-up for the fluid dynamic test case against the number of processors for a range of partition strategies using a 10,027 triangle mesh.

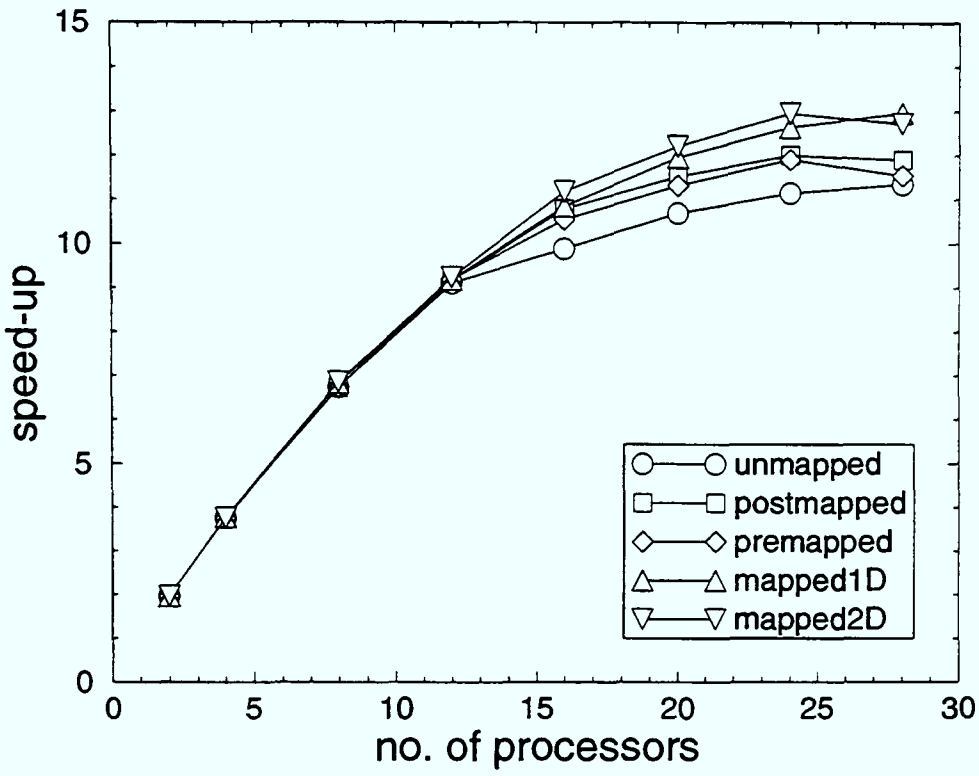

Figure 5.11: Speed-up for the fluid dynamic test case against the number of processors for a range of partition strategies using a 30,064 triangle mesh.

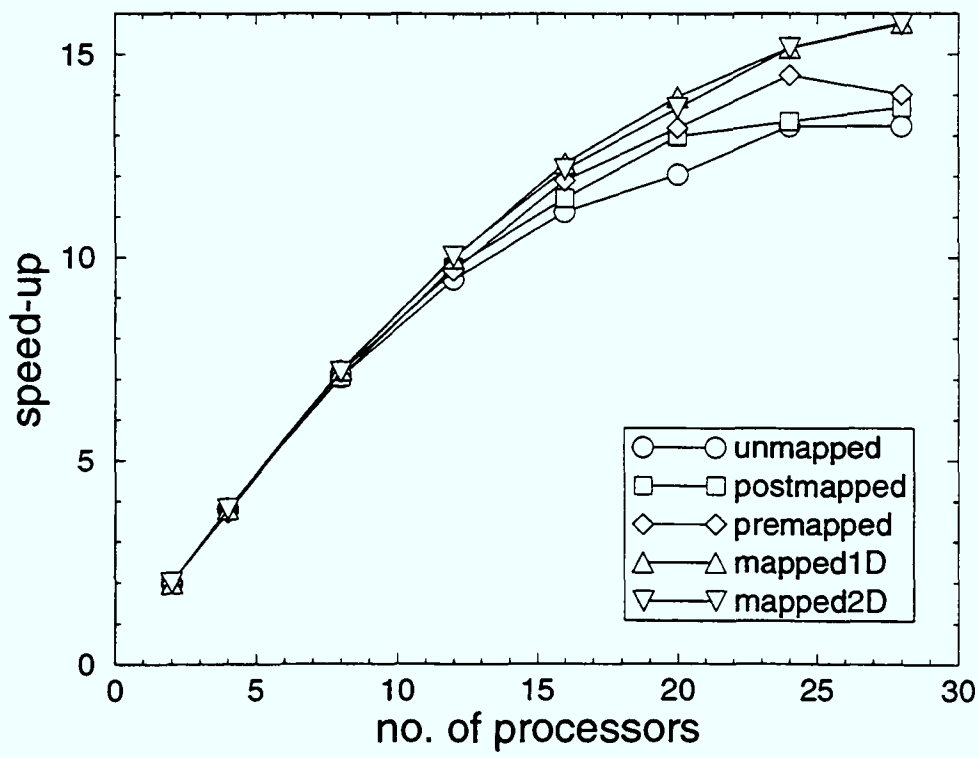

Figure 5.12: Speed-up for the fluid dynamic test case against the number of processors for a range of partition strategies using a 60,005 triangle mesh.

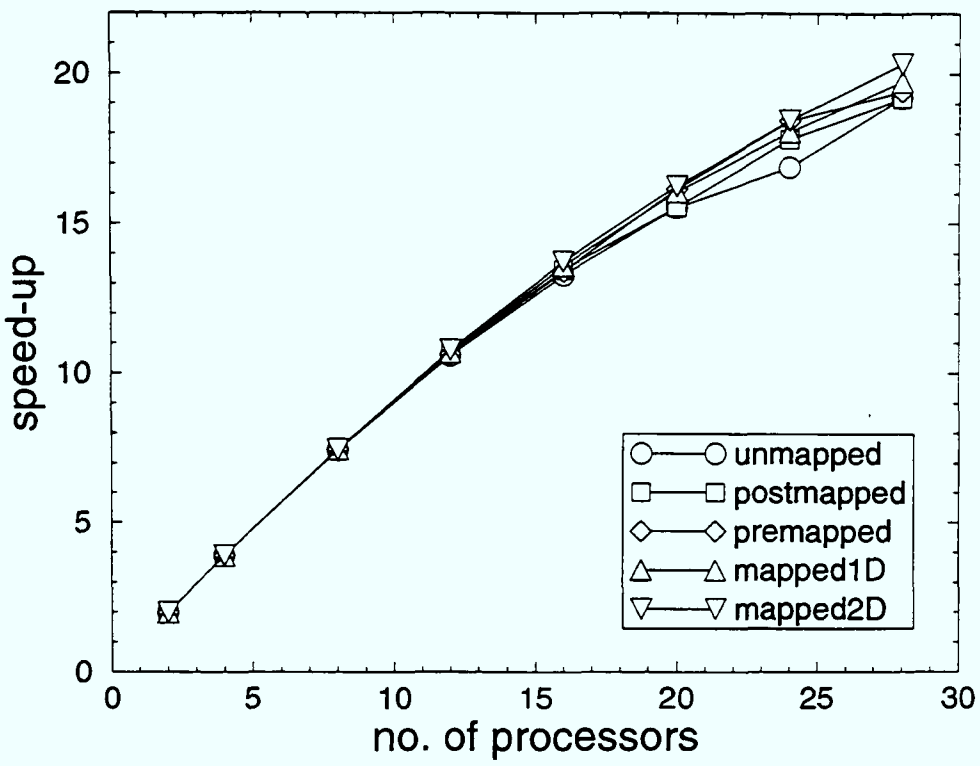

Figure 5.13: Speed-up for the fluid dynamic test case against the number of processors for a range of partition strategies using a 119,822 triangle mesh.

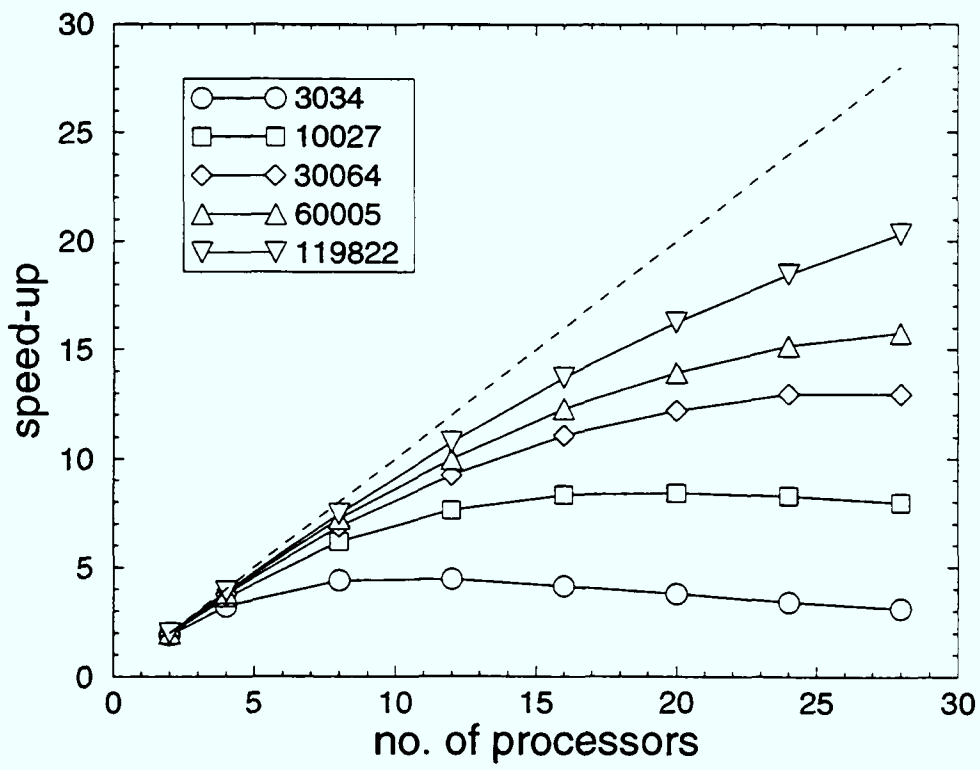

Figure 5.14: Best speed-up obtained for the fluid dynamic test case against the number of processors for a range of mesh sizes.

5.3.2 Solid mechanics test case

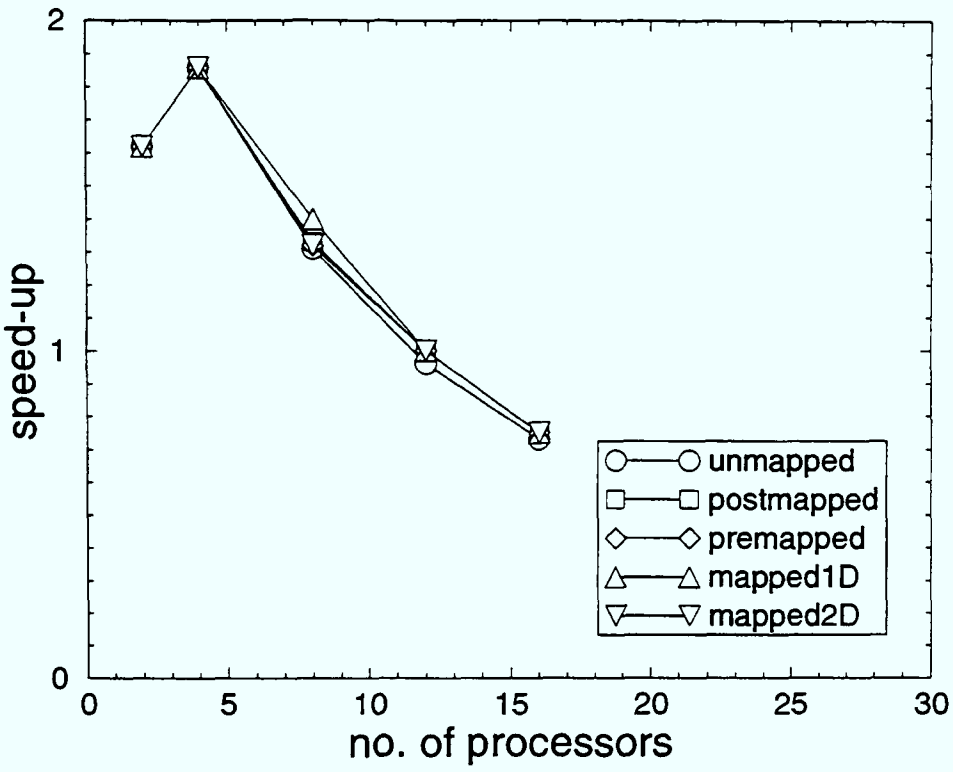

Figure 5.15: Graph of speed-up for the solid mechanics test case against the number of processors for a range of partition strategies using a 3,034 triangle mesh.

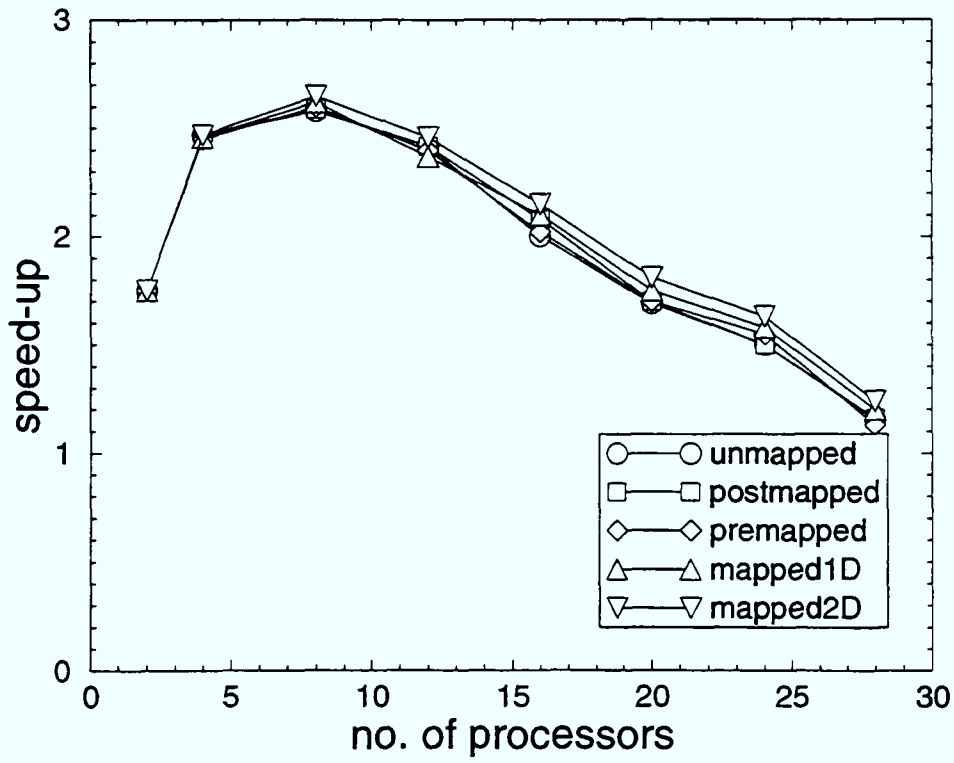

Figure 5.16: Speed-up for the solid mechanics test case against the number of processors for a range of partition strategies using a 10,027 triangle mesh.

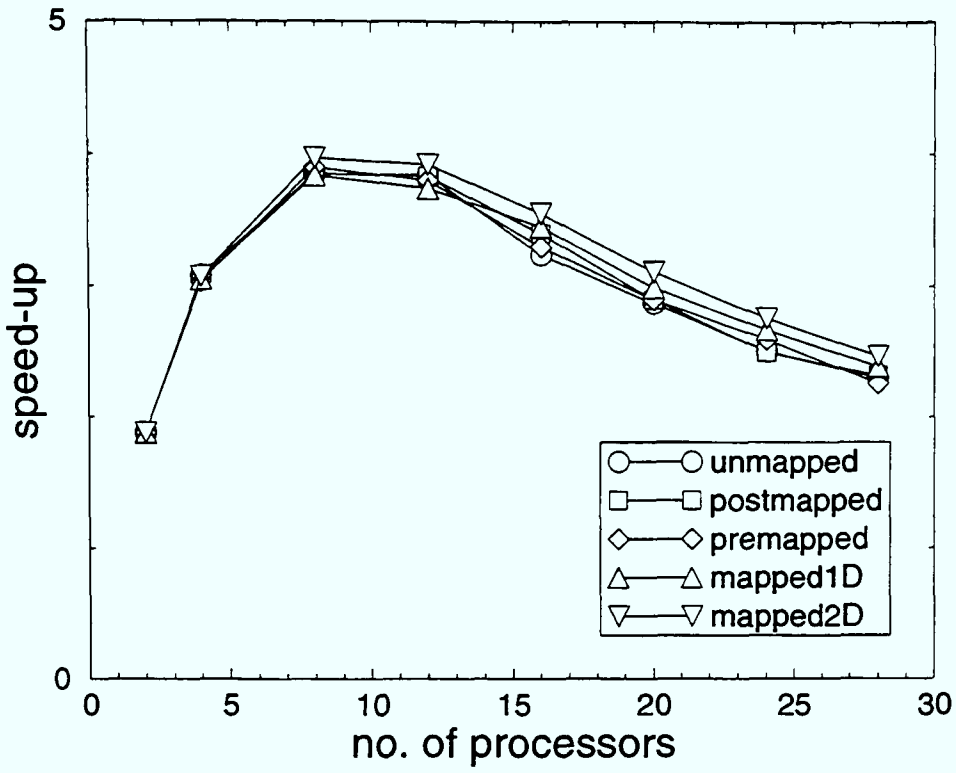

Figure 5.17: Speed-up for the solid mechanics test case against the number of processors for a range of partition strategies using a 30,064 triangle mesh.

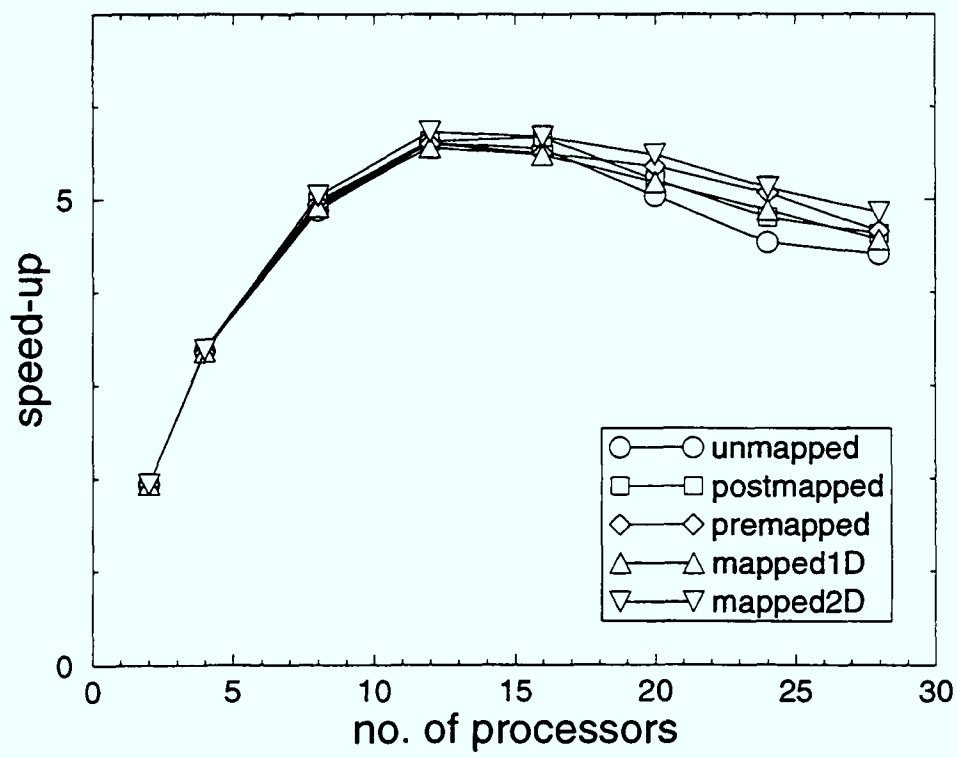

Figure 5.18: Speed-up for the solid mechanics test case against the number of processors for a range of partition strategies using a 60,005 triangle mesh.

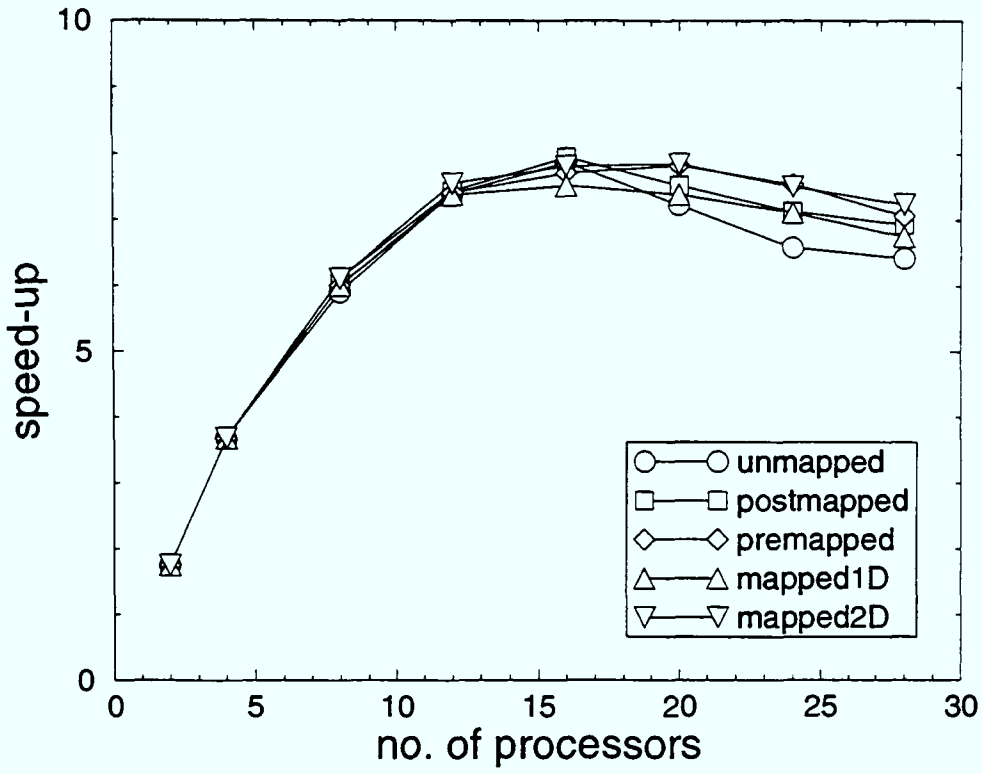

Figure 5.19: Speed-up for the solid mechanics test case against the number of processors for a range of partition strategies using a 119,822 triangle mesh.

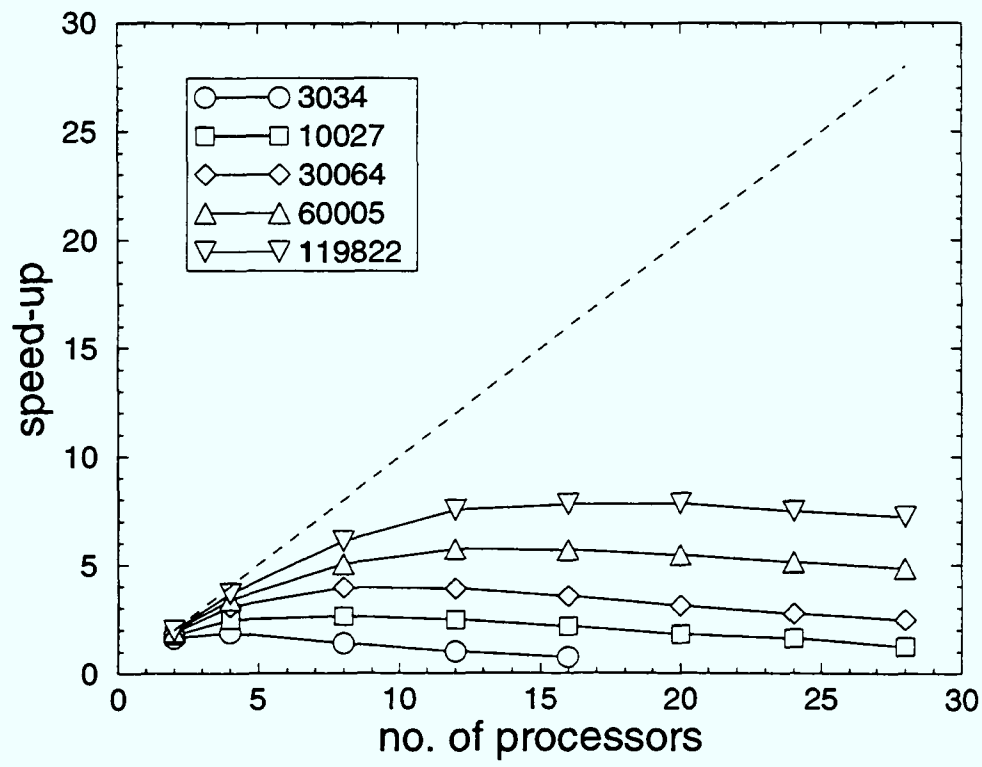

Figure 5.20: Best speed-up obtained for the solid mechanics test case against the number of processors for a range of mesh sizes.

5.3.3 Solidification test case

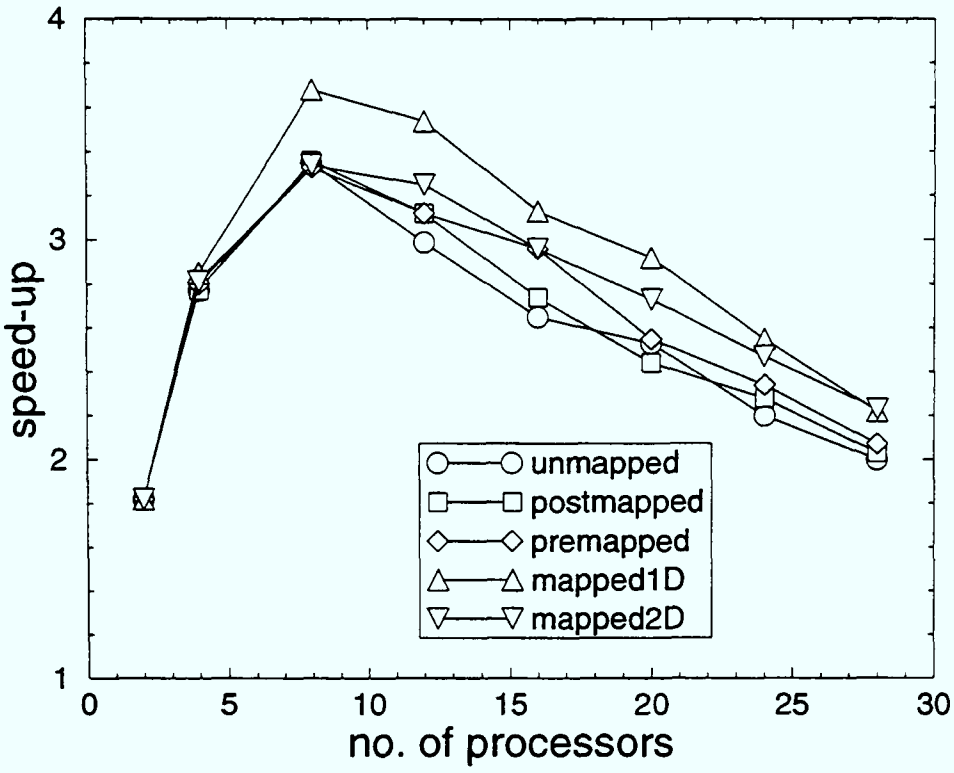

Figure 5.21: Speed-up for the solidification test case against the number of processors for a range of partition strategies using a 3,034 triangle mesh.

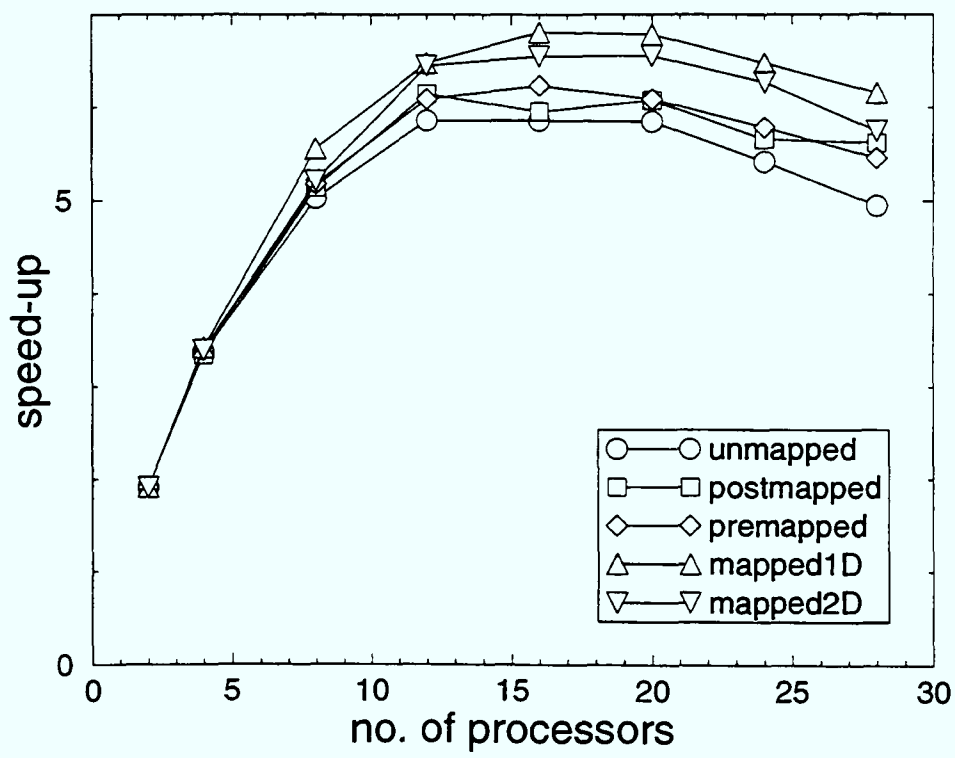

Figure 5.22: Speed-up for the solidification test case against the number of processors for a range of partition strategies using a 10,027 triangle mesh.

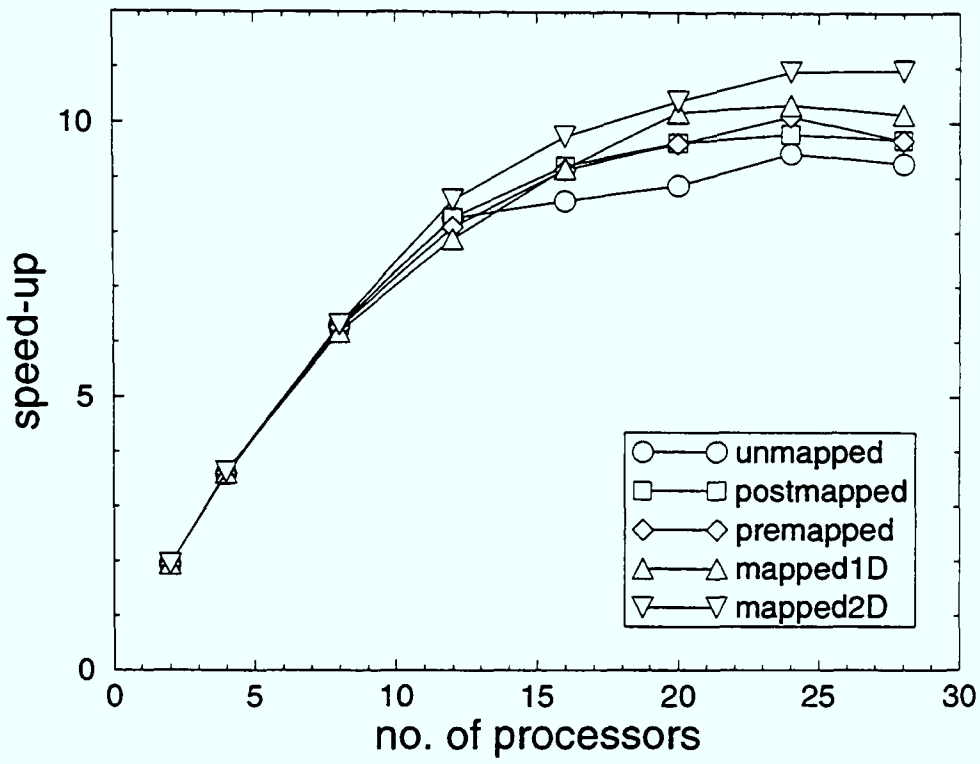

Figure 5.23: Speed-up for the solidification test case against the number of processors for a range of partition strategies using a 30,064 triangle mesh.

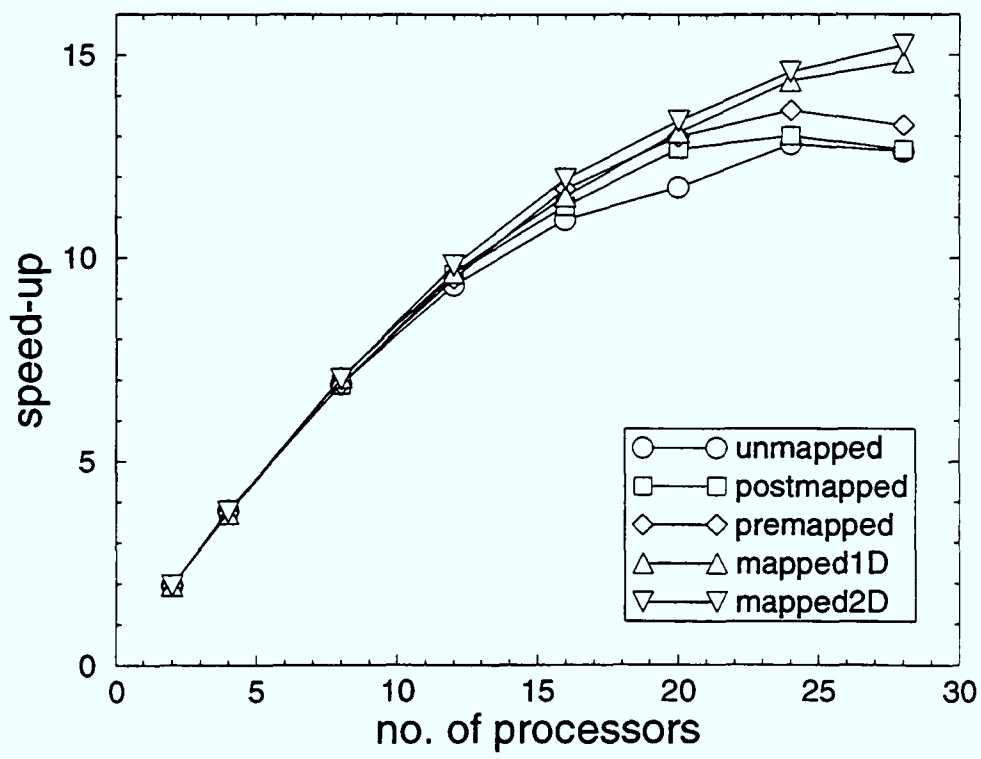

Figure 5.24: Speed-up for the solidification test case against the number of processors for a range of partition strategies using a 60,005 triangle mesh.

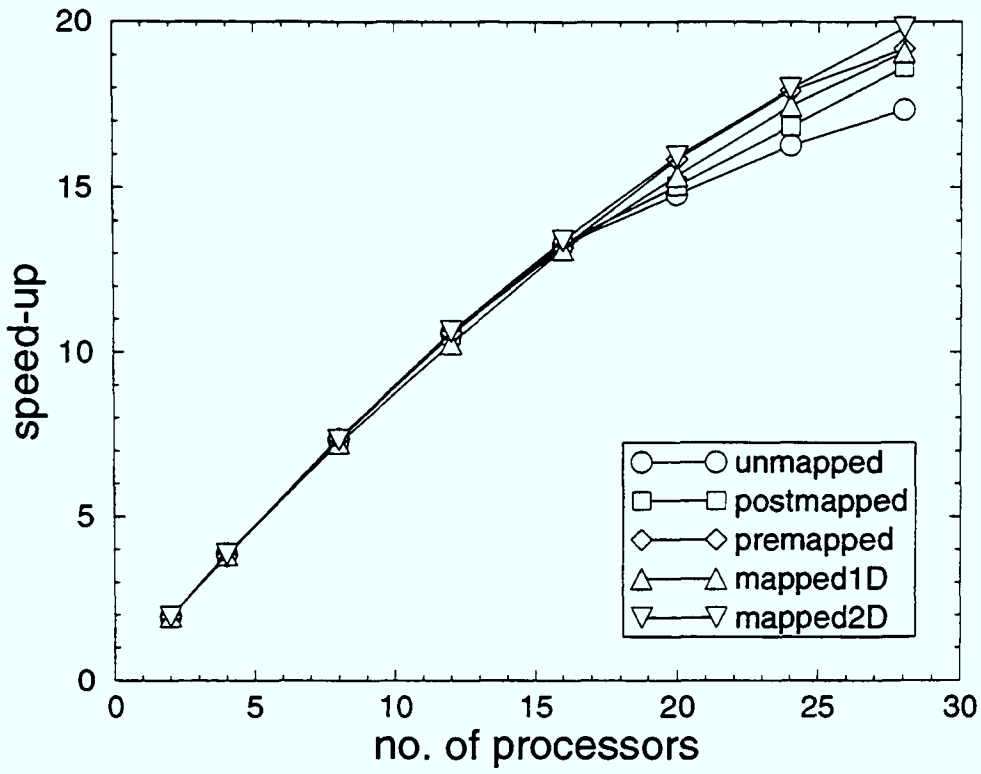

Figure 5.25: Speed-up for the solidification test case against the number of processors for a range of partition strategies using a 119,822 triangle mesh.

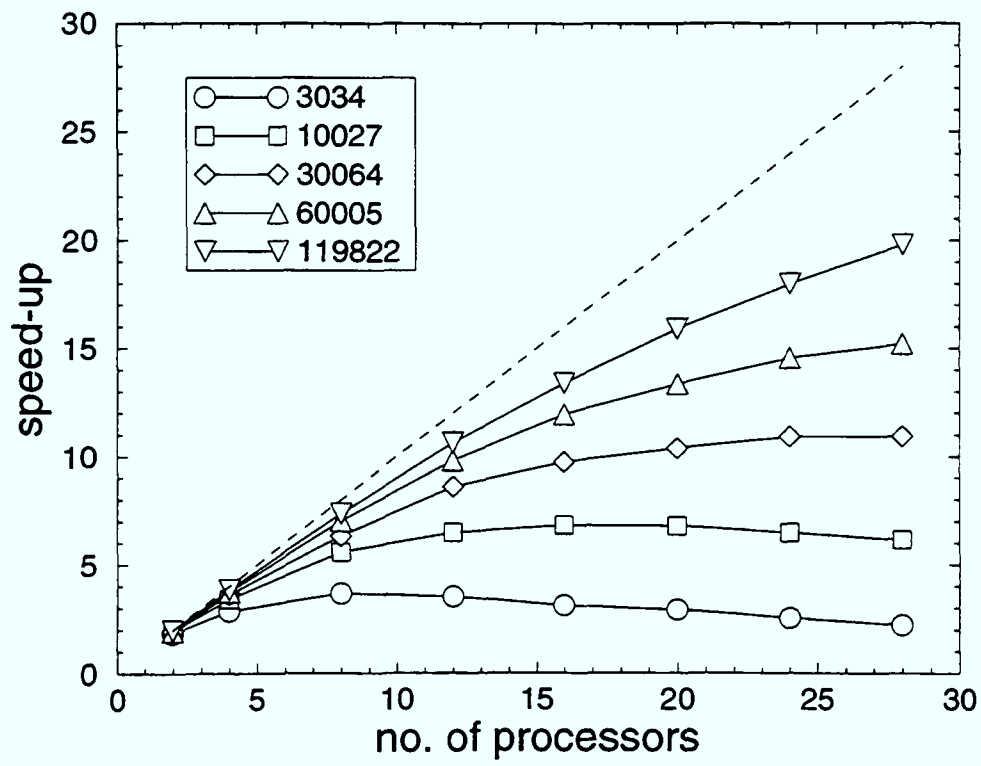

Figure 5.26: Best speed-up obtained for the solidification test case against the number of processors for a range of mesh sizes.

5.4 Improving Performance

The graphs given in Section 5.3 demonstrate a range of results from poor to good with moderate parallelism. It is fair to say that the poor results reflect poor communication performance, especially in terms of the communication start up latency. This coupled with the reasonably good calculation performance of the parallel platform, leads to a poor calculation to communication ratio. Given that a parallel machine is unlikely to ever return perfect performance all possible optimisations of the code should be sought. Two simple to implement optimisations that may be expected to realise a significant performance improvement became apparent. One is to reduce the start-up latency overhead of global commutative operations, the other is to overlap communication with calculation.

5.4.1 Latency Reduction

As communication start up latency is the dominant component of the communication overhead it seems reasonable to tackle this problem first. Profiling code execution provides a reasonably accurate view of where time is being spent in the code. For the test cases presented already in this dissertation the profiles present a clear picture of the nature of the execution. The overriding proportion of the run time was taken up in the solvers and a significant portion of that time was spent in communication. Of the time spent in communication it took very nearly the same amount of time to carry out an overlap update as it did to carry out a global commutative operation.

5.4.2 Flow and Heat Solvers

Looking closely at the Jacobi and GS-SOR solvers it becomes apparent that the preferred mode of operation in UIFS is to run these solvers to some preset maximum number of iterations, usually set at less than the amount required for convergence, and then loop over all solvers until an overall convergence criteria is reached. The logic being that no one solver should take precedence in the path to convergence. The relative importance of each component in the solution is then reflected by the number of iterations set for

each solver, e.g. 2 for each momentum, 10 for enthalpy, 20 for pressure correction. It is therefore not necessary to evaluate the residual norm at each iteration. A flag `TOMITR` in the original serial code is passed into each of the solvers to specify whether or not to run to the specified maximum number of iterations. For the test cases `TOMITR` is always true. A simple conditional test of `TOMITR` allows the norm evaluation and hence global commutative operation to be omitted. This reduces the serial run time by a small amount but has a significant effect on the parallel run time. The code for the modified Jacobi solver is given in Appendix D

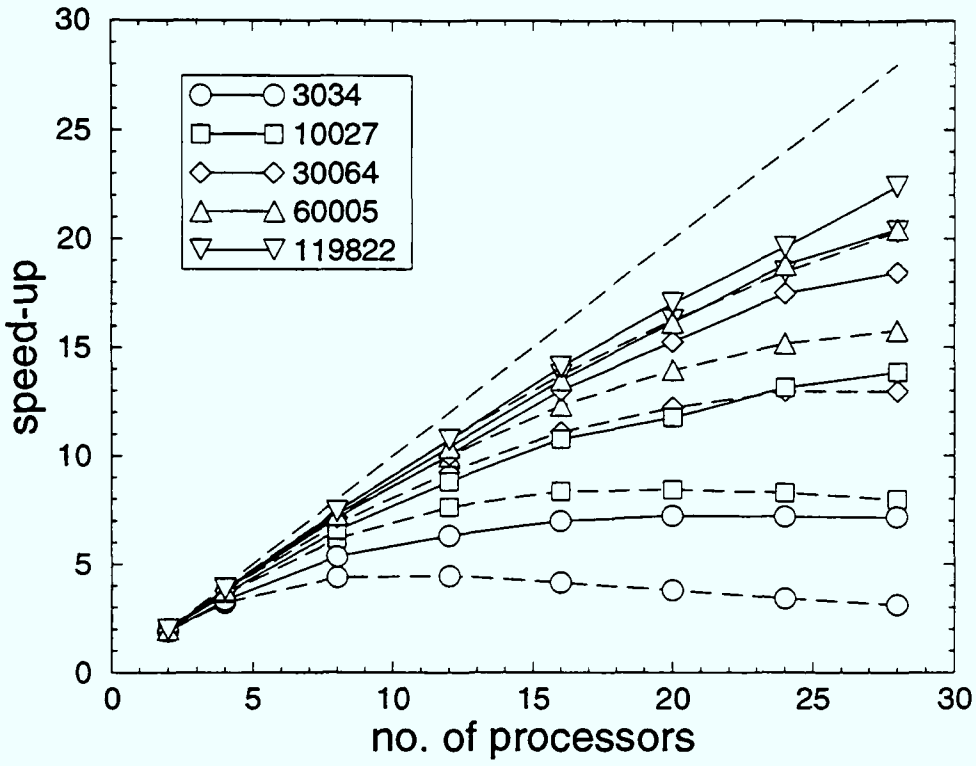

Figure 5.27: Speed-up obtained with the optimised (solid lines) and unoptimised (dashed lines) Jacobi solver for the fluid dynamics test case with a range of mesh sizes.

The effect of this modification on the fluid dynamics test case is shown in Figure 5.27. In comparison with the performance of the unoptimised solver the degree of improvement in the speed-up is more pronounced with large numbers of processors as the proportion of communication to calculation increases with the number of processors. Also the effect is more apparent with the smaller test cases as the proportion of communication to calculation is greater on the smaller, latency bound cases.

5.4.3 Solid Mechanics Solver

The conjugate gradient solver used in the solid mechanics code has two inner product operations. These operations appear in the source as two separate global summation operations. Close inspection of the code reveals that that it is possible to re-arrange the code to bring the summations to the same point in the code. Recalling equation 4.44 with the prescaled Jacobi preconditioner $\mathbf{M} = \mathbf{I}$ gives

$$\rho^{(k+1)} = \mathbf{r}^{(k+1)T}\mathbf{r}^{(k+1)} \tag{5.5}$$

Substituting $\mathbf{r}^{(k+1)}$ from equation 4.42 gives

$$\rho^{(k+1)} = (\mathbf{r}^{(k)} - \alpha^{(k)}\mathbf{u}^{(k)})^T(\mathbf{r}^{(k)} - \alpha^{(k)}\mathbf{u}^{(k)}) \tag{5.6}$$

Expanding gives

$$\rho^{(k+1)} = \mathbf{r}^{(k)T}\mathbf{r}^{(k)} + \alpha^{(k)2}\mathbf{u}^{(k)T}\mathbf{u}^{(k)} - 2\alpha^{(k)}\mathbf{r}^{(k)T}\mathbf{u}^{(k)} \tag{5.7}$$

$$\rho^{(k+1)} = \rho^{(k)} + \alpha^{(k)}(\alpha^{(k)}\mathbf{u}^{(k)T}\mathbf{u}^{(k)} - 2\mathbf{r}^{(k)T}\mathbf{u}^{(k)}) \tag{5.8}$$

Now calculation of $\rho^{(k+1)}$ requires two inner products rather than one but no longer requires $\mathbf{r}^{(k+1)}$ and so may be moved forward in the scheme to the same point as evaluation of $\alpha^{(k)}$. Consequently the three global summations necessary for the three inner products may be merged into one communication. This is similar to the work of D'Azvedo *et al* [DER93] but involves no algorithmic modification whatsoever and so has no effect on stability or convergence of the method. The time required for a global summation t_{gs} is dominated by the communication start up latency and so the time for three merged global summations is approximately equal to the time required for a single global summation. This modification is trading the time required for an inner product t_{ip} against the time required for a global summation. Remembering that that t_{gs} increases with increasing P and that t_{ip} decreases with increasing P then with increasing P there rapidly comes a point where this modification is beneficial. The code for the modified CG solver is given in Appendix D The effect of this modification on the solid mechanics test case is shown in Figure 5.28. These results use the one processor run time for the faster

unmodified CG solver to give a correct evaluation of the speed-up. What is immediately apparent from Figure 5.28 is the improvement across a range of test case sizes for four or more processors. Close examination shows that the largest test case does not show improvement until more than four processors are used. This is consistent with t_{gs} being a function of P only however t_{ip} is a function of both P and problem size N. Further increases in problem size would be expected to more clearly reveal this effect.

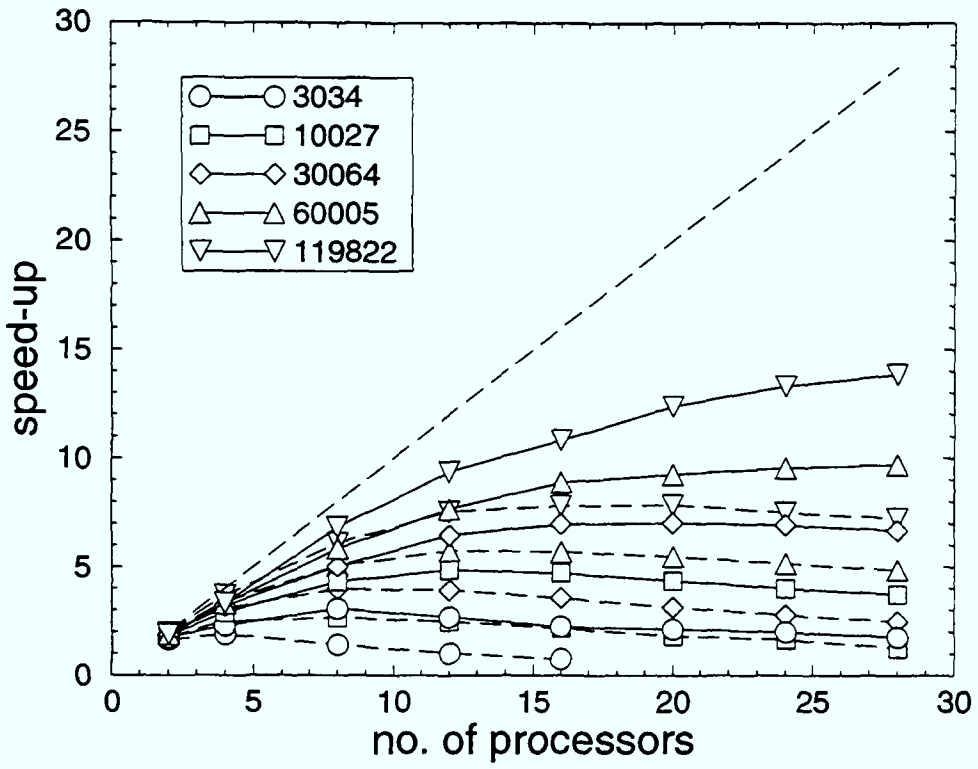

Figure 5.28: Graph of speed-up obtained with the optimised (solid lines) and unoptimised (dashed lines) conjugate gradient solver for the solid mechanics test case with a range of mesh sizes.

Figure 5.28 represents a significant improvement on the speed-up results for the unoptimised solver but the one remaining commutative operation remains an undesirable overhead. This prompts a closer examination of the global summation operation. A global summation operation has a great deal of parallelism as each processor evaluates its own partial sum. The original global summation algorithm was developed before the virtual channel router provided all to all communication. For this reasons the global summation operates in a chain fashion where each processor number p receives a sum from processor $p+1$, adds its own partial sum and passes the result to processor number

$p-1$. After $P-1$ messages processor 1 has the global summation that can be broadcast to all processors, this will therefore involve $2(P-1)$ latencies overall. So the latency overhead increases with the number of processors as discussed in section 4.2.4. This scheme ensures that each processor ends up with an identical copy of the global sum regardless of rounding errors. It should be remembered that parallel summation such as this is an order dependent calculation that will not identically reproduce the rounding errors as the number of processors varies. What is vitally important however is not that the rounding errors are the same for different numbers of processors but that the result on each processor is identical. This criteria is satisfied by a hypercube based scheme where processor pairs exchange their cumulative partial summations. This scheme also gives the lowest possible number of latencies $2n$ where $2^n \geq P > 2^{n-1}$. So for example for P between 33 and 64 only 12 latencies are required

The effect of this modification on the solid mechanics test case is shown in Figure 5.29.

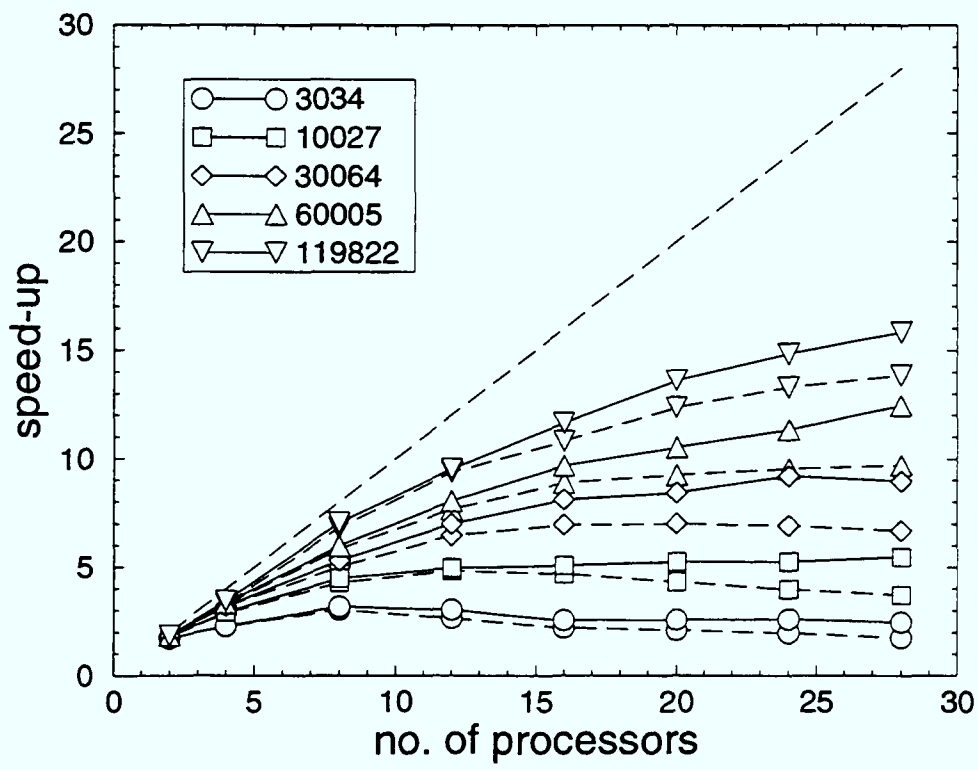

Figure 5.29: Speed-up obtained with the optimised conjugate gradient solver using a hypercube (solid lines) and a pipeline (dashed lines) global commutative for the solid mechanics test case with a range of mesh sizes.

It is apparent from the results in Figure 5.29 that the effect of the hypercube commutative is highly significant. This confirms the proposition that communication start up latency is an overridingly important factor in the achieved performance of a parallel system.

5.4.4 The Effect of Optimised Solvers on the Solidification Test Case

Figure 5.30 shows the effect of the optimised solvers and global commutative functions on the solidification test case. The reduction of latency based communication overheads in the optimised solvers has had three important effects. Comparing Figure 5.30 with the graph in Figure 5.24 for the unmodified code clearly shows the effects. Firstly, the overall level of speed-up has increased, speed-up that was in the range 12–15 for 28 processors has increased to 15–21. Secondly, the separation of the performance from the different partitions is more pronounced. Most noticeably the lines for the mapped1D and mapped2D partitions have separated, this is a direct consequence of bandwidth becoming more relevant as the latency is reduced in the solvers. The mapped1D partition has a larger amount of data to communicate and fewer communications than the mapped2D partition. Thirdly, the gradient of the mapped2D partition line is much steeper in Figure 5.30. Further speed-up could therefore be expected if more processors were available.

5.4.5 Asynchronous Communication

Many parallel platforms provide asynchronous or non-blocking communication calls to allow calculation to overlap communication. This allows subroutines to initialise a communication and return from the subroutine call before completion of the communication. The communication can then be tested for completion (synchronised) at some future point in the code. In an ideal case, unrelated code can be executed immediately after an asynchronous communication call and synchronisation effected prior to the point at which the communicated data is used. This allows the execution of unrelated code to be overlapped with the communication. Often this is not possible since the communi-

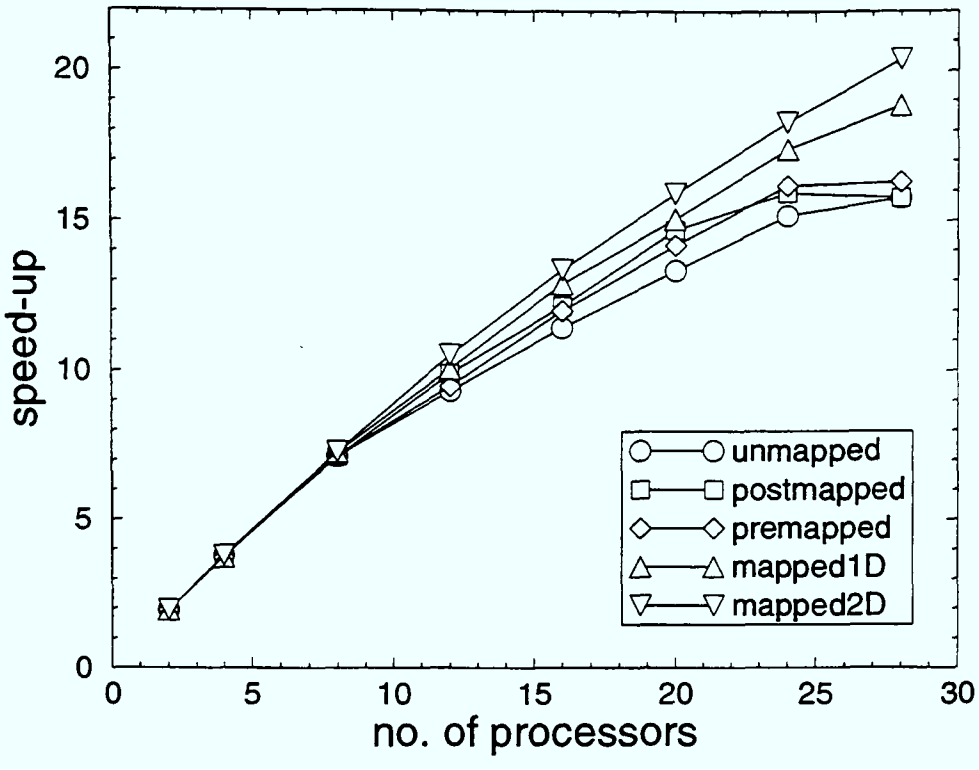

Figure 5.30: Speed-up obtained with the optimised solvers for the solidification test case with a range of partition strategies using a 60,005 triangle mesh.

cated data is immediately required. This is the case with PUIFS, however asynchronous communication can be exploited within the solvers by splitting the computation into two parts. The Jacobi and Gauss-Seidel solvers firstly solve for the variables around the perimeter of the sub-domain that are required in the overlaps of the neighbouring sub-domains. Once the perimeter calculation is complete, asynchronous communication of these variables is initiated. This leaves the time required to solve for the variables in the rest of the sub-domain (independent variables) for the asynchronous communication to complete. Completion of the communication is tested at a synchronisation point before proceeding to the next iteration. The conjugate gradient solver operates in a similar manner splitting two loops so that calculation of u and p over the independent grid points is overlapped with the communication. These schemes amount to a renumbering of each sub-domain core so that entities that are required by the overlaps of neighbouring sub-domains are numbered before the rest of the core. Such renumbering is generally acceptable as partitioning has already changed the original numbering which was often merely a consequence of the mesh generation in the first instance (Jacobi and CG meth-

ods are order independent anyway). The effect of this renumbering scheme on the mesh

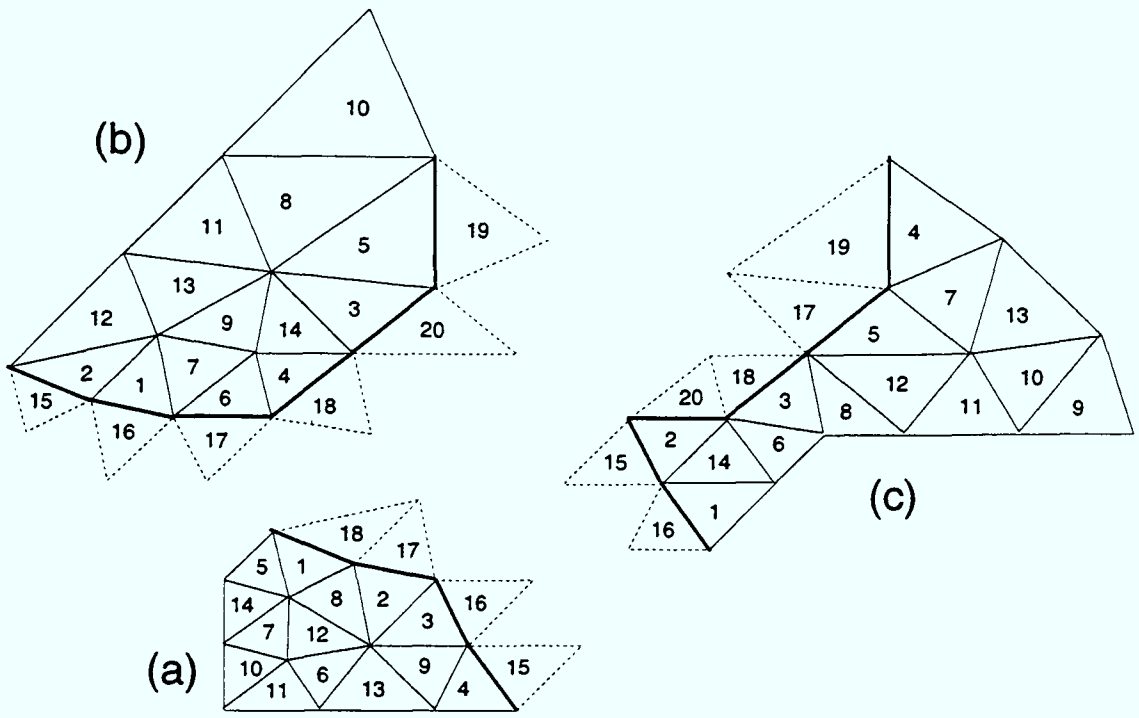

Figure 5.31: Mesh of 42 triangular elements partitioned into three sub-domains renumbered for asynchronous communication.

of 42 triangles illustrated in Figures 4.6 and 4.7 is shown in Figure 5.31. Two changes in the numbering are apparent. Firstly, the overlaps have been numbered so that overlap elements that are owned by the same sub-domain are numbered consecutively. This allows an overlap exchange to write the received overlap variables directly into memory without the need to unpack a buffer. Secondly, the elements within each sub-domain that are overlap elements on neighbouring sub-domains have been numbered before the rest of the sub-domain. Figure 5.32 shows the effect of the renumbering on the overlap communications. In contrast with the matrices shown in Figure 4.9 the communications now originate in the first few rows of each sub-domains matrix. In the iterative solver these rows are evaluated first and then the asynchronous communication of overlaps is initiated. Evaluation of the remainder of the rows in matrix equation can then proceed for that iteration while the communication is being carried out. Completion of the communication is tested before continuing on to the next iteration.

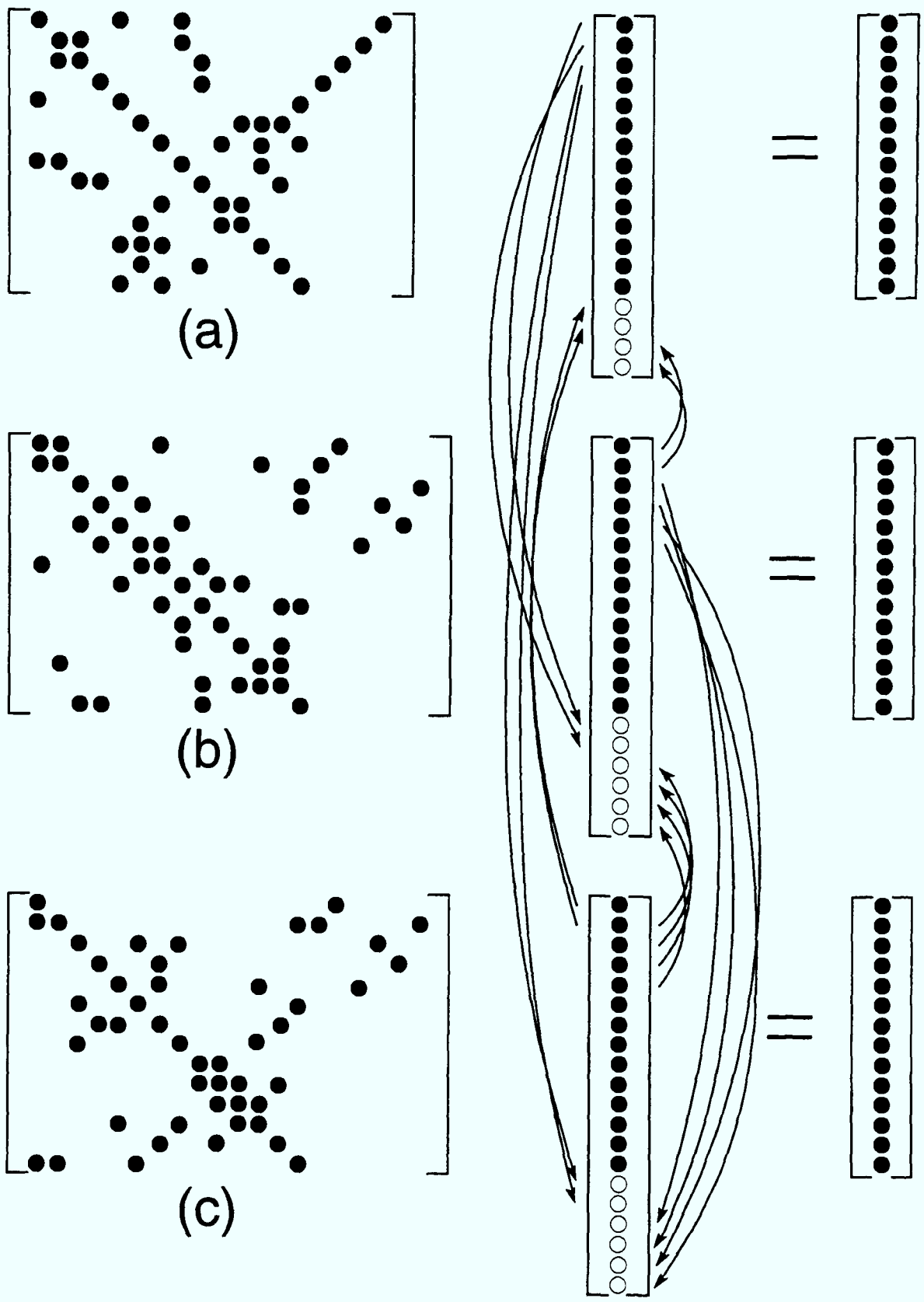

Figure 5.32: Matrices for the 42 element mesh partitioned into three sub-domains renumbered for asynchronous communication.

On the Transtech Paramid asynchronous communication is achieved through exploitation of the T800 co-processors to manage the communication. For workstation networks, notorious for their high latency, this is effected through communication buffers. The results of using asynchronous modified solvers for the fluid dynamic test case and the solid mechanics test case are presented in Figures 5.33– 5.34. Here the improvement

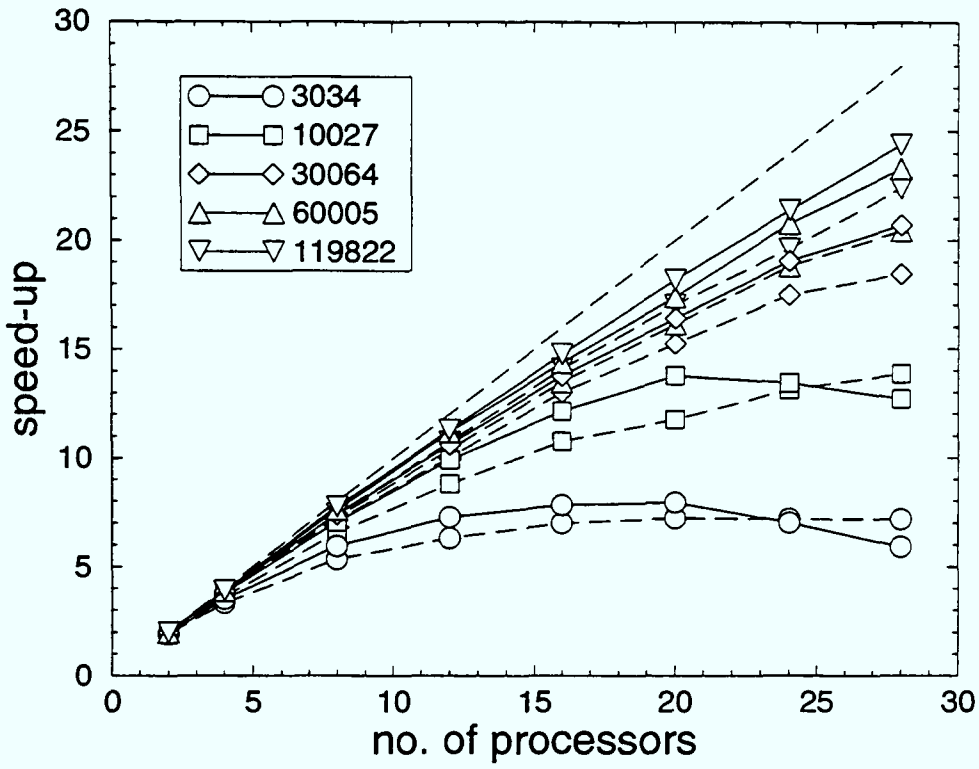

Figure 5.33: Speed-up obtained with the asynchronous (solid lines) and synchronous (dashed lines) optimised solvers for the fluid dynamic test case with a range of mesh sizes.

in performance over the synchronous results is clear. These results paint a very different picture of parallel performance on a Transtech Paramid than for instance Figure 5.15. These results reinforce the assertion that parallel performance is highly code, problem and machine dependent. Overlapping the communication with calculation through the use of asynchronous message passing has effectively concealed the bandwidth requirement for communication of overlap data. That is providing that there is enough calculation to conceal the communication. The curves for the 3,034 and 10,027 element test cases in Figure 5.33 show a drop in performance in comparison with the synchronous results for 28 processors. With large P and a small problem the amount of calculation may not

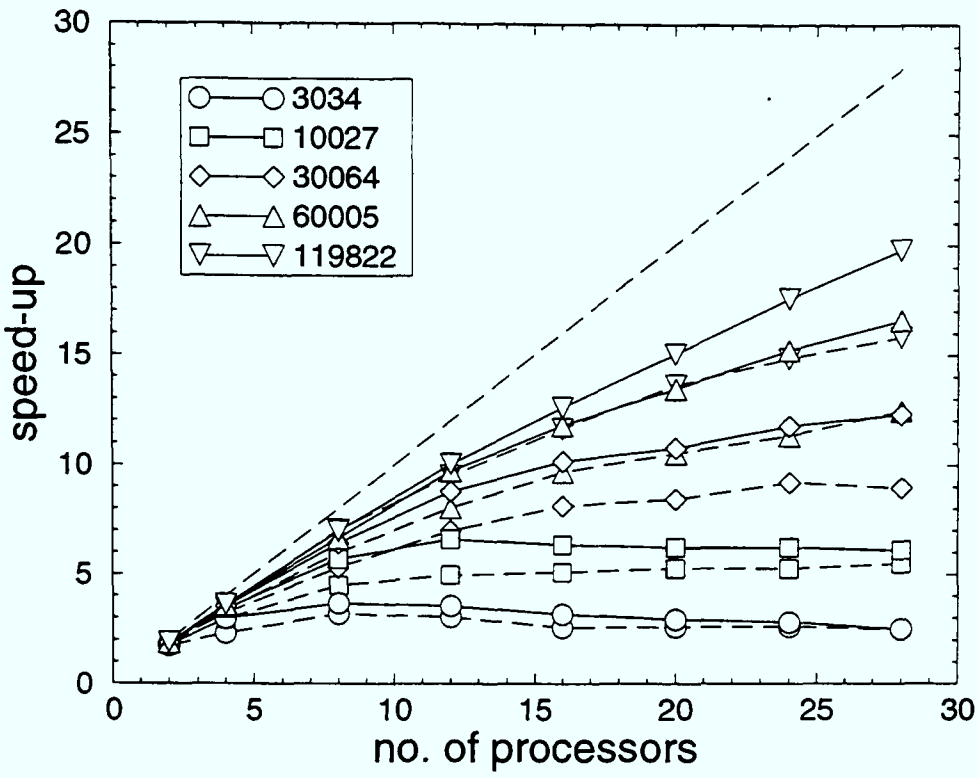

Figure 5.34: Speed-up obtained with the asynchronous (solid lines) and synchronous (dashed lines) optimised solvers for the solid mechanics test case with a range of mesh sizes.

be sufficient to overlap all of the communication. Figure 5.35 clearly shows how effective this hiding is for the 60,005 element solidification test case. Here the spread in performance between the partitioning strategies is far less apparent than the synchronous case in Figure 5.30. The mapped1D and mapped2D partitions still have a performance advantage but the performance from the other partitions is now comparable with the mapped partitions. The premapped, postmapped and unmapped partitions now look capable of returning further speed-up beyond the 28 available processors, which is clearly not the case in Figure 5.30. The mapped1D and mapped2D partitions return near identical performance as the bandwidth overhead of the mapped1D partition is effectively concealed and the advantage of the lower latency requirement for the 1D partition becomes significant. These results invite investigation of the performance beyond 28 processors. There must inevitably come a point at which the performance returned from the different partitions becomes more significant. Eventually the amount of computation in the each sub-domain core will not be sufficient to fully overlap the communication.

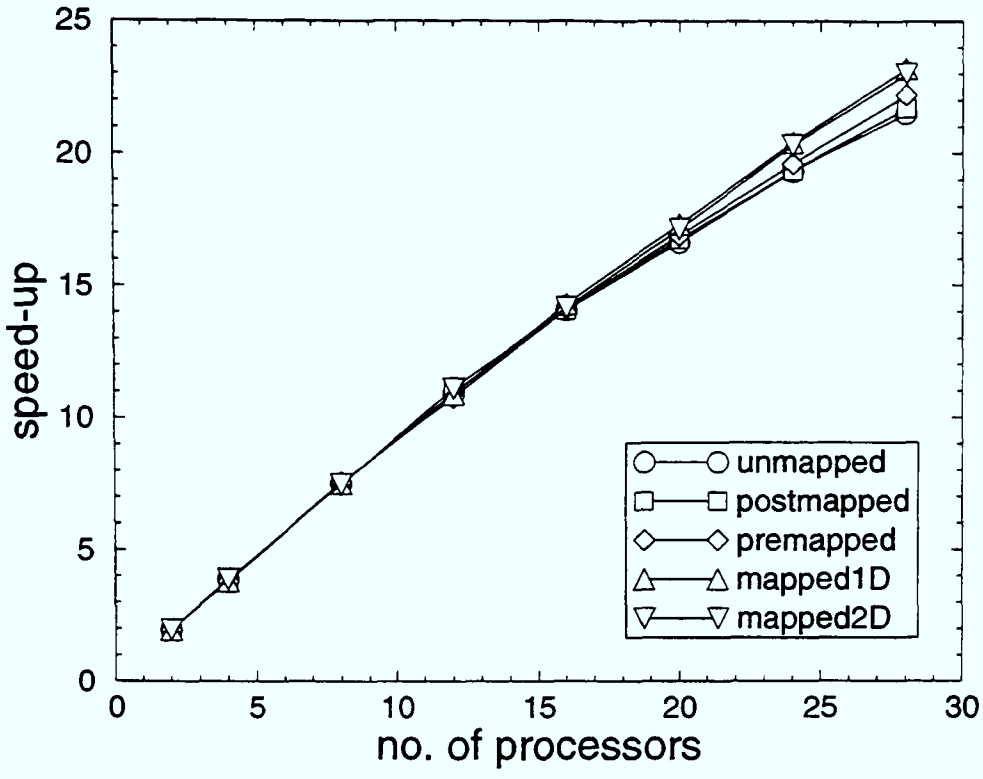

Figure 5.35: Speed-up obtained with the asynchronous optimised solvers for the solidification test case with a range of partition strategies using a 60,005 triangle mesh.

Code for the asynchronous Jacobi and conjugate gradient solvers is given in Appendix E

5.5 Summary

Speed-up has been used throughout for the presentation of these results. If larger numbers of processors were available the temptation to run still larger test cases may make speed-up curves impractical. However it is felt that speed-up graphs present a more clear picture of parallel performance than efficiency or run time graphs. To put the performance of the i860 processors in context, the execution of UIFS on a single i860 PE is approximately 10% faster than the run time on a state of the art Sun Sparc20 75MHz processor. The results presented in this chapter have shown how the communication overhead in parallel processing is the limiting factor to achievable performance. The precise nature of the communication must be addressed and optimised if an acceptable performance is to be obtained. Acknowledgement of the topology of the machine in the mesh partition has been shown to be of significant importance. There comes a point how-

ever when all possible optimisations have been applied and yet the performance remains disappointing, as in the small test cases in Figure 5.34. The only remaining significant factor is start-up latency which is unavoidable *Communication start-up latency must be reduced if a parallel machine is to achieve computational scalability.*

Chapter 6

Automation of Parallelisation

Parallelisation of a large unstructured mesh CM code is a time consuming, error prone and labour intensive task. Any tool that can help to alleviate the problem of parallelisation will be a welcome asset. Some success has been shown with environments and libraries for the authoring of parallel unstructured mesh codes but this is of no help to the parallelisation of existing codes and should not be forced upon code authors who have little or no interest in parallelism. Much success has been shown with the Computer Aided Parallelisation Tools (CAPTools) for automation of the parallelisation of structured mesh codes [JCI+94]. Extension of CAPTools to provide automation of the parallelisation of unstructured mesh codes presents some new and interesting problems.

6.1 Computer Aided Parallelisation Tools

CAPTools is an interactive toolkit for Computer Aided Parallelisation of mesh based FORTRAN codes. The objective of CAPTools is to automate as much as possible of the process of parallelising mesh based numerical FORTRAN codes. The principle axiom of CAPTools is to generate code of equivalent or better quality to that which can be produced manually (Perhaps with minimal user interaction). The code generation techniques employed therefore match those used successfully for numerous manual parallelisations. Parallelisation relies on an accurate analysis of the target code achieved

through the use of sophisticated interprocedural symbolic algebra techniques to analyse the code and produce an accurate dependence graph that can be enhanced with knowledge supplied by the toolkit user. Parallel code is generated based on a data partition constructed using the dependence graph allowing execution control and communication requirements to be subsequently identified. An X windows based graphical user interface provides an environment for the user to navigate the code, visualise dependencies and *interact* with the knowledge base throughout the parallelisation process.

6.1.1 Dependence Analysis

Dependence analysis builds a directed graph $D(S, R)$ where the nodes S of the graph are executable statements and the edges R represent the relationship of required execution order (the dependencies). There are four basic dependence categories;

True dependence - resulting from the data flow between a source (point of assignment) and a sink (point of use).

Control dependence - when execution is controlled by a conditional statement.

Anti-dependence - caused by re-assignment of a used variable in a sink statement.

Output dependence - where a source variable is re-assigned the order of assignment must be maintained.

The location of a dependence is also significant, does the dependence exist between loops or within a single iteration of all surrounding loops? Dependence analysis is achieved using the Greatest Common Divisor test (GCD), the Bannerjee tests [Ban79, Ban88] and the Symbolic Inequality Disproof Algorithm (SIDA) [Joh92] The graph is then pruned using all previous dependence information to give a precise representation of the dependence structure of the code.

6.1.2 Data Partitioning

Distribution of a programs data over a parallel machine begins with the determination of the set of variables that are to be distributed and the way in which they are to be distributed. Two techniques are currently employed. One is to select an array index that can be partitioned. The chosen array should be a significant component inside the dominant (most compute intensive) loop identified through profiling of the code execution. The other technique is to select a loop (again usually the most dominant) and partition all arrays contained within the loop in accordance with their use of the loop counter. For example;

```
DO I = 1, NI-1
  DO J = 1, NJ-1
    DO K = 1, NK-1
      V(I,J,K) = (X(I+1)-X(I)) * (Y(J+1)-Y(J)) * (Z(K+1)-Z(K))
    END DO
  END DO
END DO
```

Selecting the inner `K` loop as the partitioned loop will partition `V` in its third index and `Z`. Other array variables that are assigned or used by a partitioned array will inherit the partition if a linear relationship exists involving the index expression in relation to the already defined array partition. So the statement;

```
Q(K,J,I) = V(I,J,K)
```

will imply that the partition of the third index of `V` is inherited as the first index of `Q`. Inheritance propagates the partition via the dependencies to partition as many arrays as possible in all routines of the code (interprocedural). The partition is implemented through the introduction onto each processor of lower `CAP_L` and upper `CAP_H` limits to the index of the array. So a declaration that was

```
INTEGER   NI, NJ, NK
INTEGER   V(1:NI,1:NJ,1:NK)
```

can become in parallel

```
INTEGER   NI, NJ, NK
INTEGER   CAP_L, CAP_H
INTEGER   V(1:NI,1:NJ,CAP_L:CAP_H)
```

These CAP_ variables hold different values on each processor. The values are calculated on all processors at run time as functions of the assigned range of the partitioned component of the array along with the number of processors employed and the number of the processor on which they are being calculated.

6.1.3 Execution Control

Execution control masks are used to enforce an 'assign only allocated data' rule to the data partition. These are a conditionals that determines whether a statement should execute on a particular processor. The control mask can take the form;

```
IF ( CAP_L .LE. <expression> .AND. <expression> .LE. CAP_H ) THEN
```

where the partition of an array has inferred the mask. This can be propagated to the loop limits so the previous example can become

```
DO I = 1, NI-1
  DO J = 1, NJ-1
    DO K = MAX(1,CAP_L), MIN(NK-1,CAP_H)
      V(I,J,K) = (X(I+1)-X(I)) * (Y(J+1)-Y(J)) * (Z(K+1)-Z(K))
    END DO
  END DO
END DO
```

Execution control masks are propagated using dependence information to cover as many statements as possible to maximise parallelism. Any statement that is not masked must be executed on every processor.

6.1.4 Communication

Having partitioned the arrays and set masks to control the execution, the use of data that is not on the assigning processor can be determined. Communication is requested by a reference within a statement to access data that is not on the processor executing

the statement. Determination of the communication involves comparison of the execution control mask of a statement with the location of the data as defined by the data partition. Both are specified in terms of the partition range variables which are only assigned values at run time. Calculation of the communication must therefore be based on symbolic inequalities involving the variables. Communication requests are then migrated to as early a point in the code as is legal and profitable, often exiting loops to allow bulk communications. Barriers to movement are detected from true dependencies indicating the location of assignments of the data to be communicated, either in an earlier code section or in an earlier iteration of a surrounding loop. Several requests for communication of the same or subsets of the same data can be generated and migrated to the same place. These requests can often be merged into a single communication.

In the above example, the reference to `Z(K+1)` will generate a communication request for the single value `Z(CAP_H+1)` which can be migrated out of the code section and possibly merged with similar requests.

A special case exists for commutative operations which can exploit parallelism where it appears to be prohibited by a loop carried true dependence.

```
DO I = 1, N
  P = <function>(P,R)
END DO
```

Where the function may be max, min, + or $\times$.

```
DO I = 1, N
  IF ( <function>(P,R) )  P = R
END DO
```

Where the function may be $<$, $\leq$, $=$, $\geq$ or $>$.

6.2 Generic Parallelisation Methods for Unstructured Mesh Codes

In seeking to automate the parallelisation of any unstructured mesh based CM code the methods used must be sufficiently generic to cope with the diversity of code structures.

The ideas described in this thesis have been demonstrated on UIFS, a large two dimensional FV element based code integrated with a FV node based code. The similarities between the FV scheme and the more popular FE schemes are such that the two present much the same case for parallelisation. Extension of the strategies into three dimensions is of little consequence, the techniques used are to a large extent dimensionally independent. The strategies described in this thesis have been successfully applied to other complex codes, for example the aerodynamics code SAUNA [PS92, IFB95] which is a three dimensional multigrid block structured and unstructured mesh Euler and Navier Stokes code. At a conceptual level the strategies are certainly of general application to unstructured mesh codes with localised data dependencies. Long range or global dependencies would require large overlaps and consequently large amounts of data to be communicated between processors. Other strategies than those described in this thesis will be required to parallelise such codes.

This thesis has described the strategy of *geometric* domain decomposition. The geometric nature of the code and its data structures is understood and acknowledged throughout. Geometry (topology) is used as a basis for the partitioning of the mesh, the construction of overlaps and the scheduling of messages. Geometric and topologic concepts are convenient as models for human understanding but are of limited use as models for a machine to understand and operate upon. The data structures used in a code must be treated in more abstract terms if an automated analysis is to have any success. Partitioning based on a graph has now been demonstrated to be a practical generic method to obtain a partition of the problem space. Similarly the derivation of secondary partitions given the relationship between primary and secondary mesh entities is a generic operation. Some of the open problems are;

- How is the graph to provide the primary partition obtained?
- How are the relationships to other mesh entities obtained?
- How are the communication requirements determined?

- How are the sub-domains renumbered?
- How are the overlaps determined?

6.2.1 Application of CAPTools Structured Mesh Techniques to Unstructured Mesh Codes

The methods developed in CAPTools for dependence analysis are immediately applicable to unstructured mesh codes. Construction and pruning of the dependence graph for unstructured mesh codes presents no new problems over structured mesh codes and so is supported by the current version of CAPTools. CAPTools can identify the arrays that need to be partitioned. For example, in UIFS it can identify all element based arrays and indicate that they should have the same partition definition. Partitioning of the array index into segments defined on each processor using `CAP_L` and `CAP_H` is possible with an unstructured mesh code but unlikely to give good results as the ordering of arrays is unlikely to reflect the structure of the mesh. A more suitable method for an unstructured mesh code is to define the partition with an array indicating the owning processor for each individual entity allowing the flexibility of an efficiently mapped partition as demonstrated in Chapter 3. Such information can only be determined at run time once the mesh structure is known and cannot be predetermined using inequality based relationships as with a structured mesh. If a set of graphs that represent the mesh can be identified from analysis of the code then array partitioning can be achieved by handing one of the graphs to a graph partitioning code such as JOSTLE. The dependencies that are identified between the graphs can then be used to derive secondary partitions for the other graphs.

Parallel execution control can be determined in the same way as for structured meshes except that the owner computes masks become functions of the partition list and the processor number (Section 3.3.3). Determination of what to communicate and where to communicate is again largely the same for unstructured mesh codes as for a structured mesh. As with a structured mesh, comparison of a statement execution control mask and the partition definition of an array that the statement accesses, can be used to

determine if a communication is required. Due to the added complication of the partition being defined as a run time calculated list of processor numbers, the implementation of communications in the generated code requires further examination.

6.2.2 Data Structures for an Unstructured Mesh

Section 3.1 describes the entities and relationships that are used to specify an unstructured mesh. The data structures used to encapsulate these relationships may be implemented in an almost limitless variety of schemes. A suitably generic means of describing data relationships is required that can accommodate not only the existing diversity of expression but also any future system that may not yet be conceived. The elements of

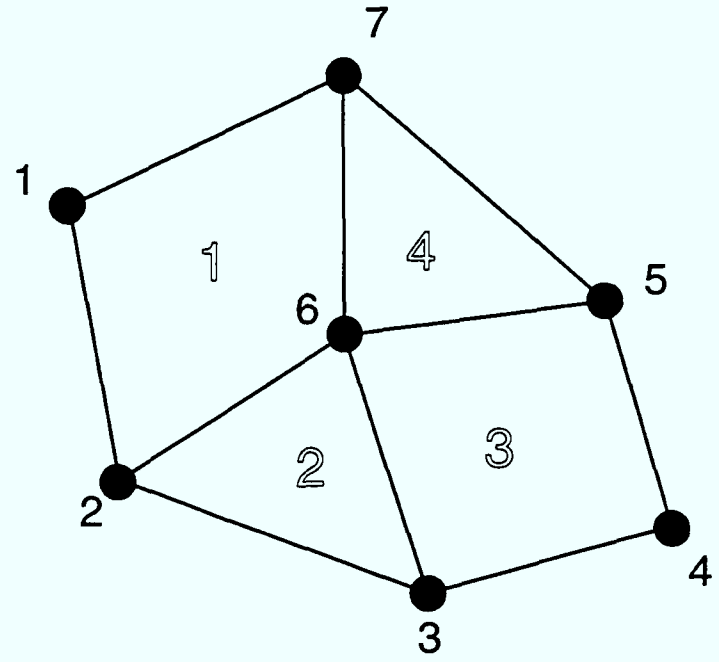

Figure 6.1: Four element mesh.

the mesh illustrated in Figure 6.1 could for example be described in terms of the grid points in any of the following three formats.

a) A two dimensional array of length number of elements and width maximum number of grid points per element.

```
1 2 6 7
2 3 6 0
3 4 5 6
6 5 7 0
```

b) A one dimensional list of length $\sum_1^n (m_n + 1)$ where n is the number of elements and m_n is the number of nodes in element number n. This lists for each element the number of nodes in the element followed by the node numbers.

```
4 1 2 6 7 3 2 3 6 4 3 4 5 6 3 6 5 7
```

c) A one dimensional list of length $\sum_1^n m_n$ listing the node numbers in each element with the last node for each element identified by being negative.

```
1 2 6 -7 2 3 -6 3 4 5 -6 6 5 -7
```

Many alternative means of expression of the relationship between grid points and elements could be conceived. A number of other variables will usually be required to complete the descriptions. Some possibilities are;

- The total number of grid points
- The total number of elements
- The number of element shapes
- The number of elements of each shape

Given that this is only one of the many relationships that may be used in the mesh description it is apparent that an almost limitless variety of description is not only possible but probable. It would be a greatly simplifying strategy to prescribe a data structure that could be used to describe a mesh in such a way that it may be parallelised. Oplus, for example, describes mesh entities as sets and the relationships between them as pointers. This elegantly simple system can express in an obvious form all that is required of an unstructured mesh. But we cannot force the re-authoring of a code in terms of alternative data structures, this runs contrary to objective (ii) and re-educating the programming community for the sake of automatic parallelisation is an impractical and unnecessary task. Parallel utilities such as those described in Section 4.2.4 must have a standard data structure that they can use to interface between the codes data structures and the utility data structures. This information is only available at run time but the means to extract the information at run time must be generated at compile time.

6.2.3 Inspector Loops

Inspector loops [vH92, MSS+88] can be generated from the code to build at run time the mesh topology as a graph of entity pairs. The following code fragment comes from the Jacobi solver listed in Appendix C.1

```
          DO 300 I = IONE, TOTELE
             X(I) = B(I)
             DO 200 J = IONE, SYSINX(HEADER,I)
                X(I) = X(I) + SYSMAT(J,I) * OX(SYSINX(J,I))
200          CONTINUE
             RESVAL(I) = ABS ( X(I) - OX(I) )
300       CONTINUE
```

Calculation of `X(I)` requires the value of `OX(SYSINX(J,I))` that is possibly on another processor. The two variables `X` and `OX` are known (from the partition definition) to be associated with the same (mesh) entity. This loop can be used as an inspector to create (or add to) a description of the dependence between the like entities in this example as a directed graph `TOPOLOGY` of entity number pairs. Only statements relating to the index expression in the array usage (in this case `SYSINX(J,I)`) and the expression in the execution control mask on the statement (in this case `I`) need to be reproduced. These expression values are stored as a pair of integers relating the requiring processor and the data owning processor.

```
          COUNT = 1
          DO 300 I = IONE, TOTELE
             DO 200 J = IONE, SYSINX(HEADER,I)
                TOPOLOGY(COUNT) = I
                TOPOLOGY(COUNT+1) = SYSINX(J,I)
                COUNT = COUNT + 2
200          CONTINUE
300       CONTINUE
```

Using routines to initialise (`INITTOP`) the `TOPOLOGY` and check for duplicates before adding to the graph (`ADDTOP`) the amount of information in the graph can be minimised.

```
          CALL INITTOP(TOPOLOGY)
          DO 300 I = IONE, TOTELE
             DO 200 J = IONE, SYSINX(HEADER,I)
                CALL ADDTOP(TOPOLOGY,I,SYSINX(J,I))
200          CONTINUE
300       CONTINUE
```

6.2.4 Partitioning

Partitioning has now been demonstrated to be successful given any graph. The inspector loops build at run time a set of directed graphs that describes the dependence between entities. Selecting the dominant graph, this graph can be undirected and sorted to remove duplicates to produce an undirected graph suitable for passing to a code such as JOSTLE.

Section 3.3.2 describes the use of rules for the determination of sub-domain overlaps. These rules are derived from a knowledge of the dependencies of the code. This is not a suitable method for automatic generation of overlaps. JOSTLE will provide a primary partition which can be used as a basis for derivation of secondary partitions as outlined in Section 3.3.1. The directed graphs returned by the inspectors can now be used instead of rules to construct overlaps onto the partitioned mesh entities using the standardised data structures that they construct.

In the previous example the topology pairs represent the element requiring a value and the element owning the required value. If `PL_X` is the processor list array for the array `X` and `PL_OX` is the processor list for `OX` then a communication is required if `PL_X(TOPOLOGY(I))` is not equal to `PL_OX(TOPOLOGY(I+1))`. The required entity number and its owning processor number are used to construct communication lists as described in Section 3.3.4. The generic utilities based on those developed for PUIFS to perform communication list construction and communication are of the form:

```
OVERLAP ( COMMS_SET_ID, TOPOLOGY, PA, PB )
SWAPOVER ( COMMS_SET_ID, VAR, LENGTH, STRIDE )
```

Where `COMMS_SET_ID` indicates a particular communication set (assigned in `OVERLAP` and used in `SWAPOVER`). `TOPOLOGY` is returned from the inspector loop immediately preceding the call to `OVERLAP`. `PA` and `PB` are the processor lists for the entities involved in the relationship. `VAR` is the variable to be communicated. `LENGTH` is the data item size (single, double precision) and `STRIDE` is the distance between consecutive entities in the `VAR` array.

6.2.5 Communication Generation

The requirement for a communication can be detected using execution control and partition definition information. To implement unstructured mesh communications, two related requests are generated, one for the communication itself and the other for the associated inspector loop representing the relationship that caused the communication. Both requests are then migrated as far as possible up the control flow of the code. Typically the inspector request will migrate further than the communication request since it is not usually a function of program solution variables, but only integer pointer arrays that are often, for example, read into the code near the start of its execution. This often allows inspectors to be executed only once for each run of the code whilst the resulting information, i.e. the communication lists, are subsequently used many times. The merging of communications requires the union or merger of related inspector loops. The resulting communication list will contain all of the information of both communication requests allowing a single generated communication to perform all required data transfer.

6.2.6 Renumbering

It is essential to pack the partitioned mesh and data in each sub-domain in order to achieve scalability of memory. Section 3.3.3 has shown that unless the partitioned problem is renumbered to a local numbering scheme then globally dimensioned pointer arrays are required to indirect addresses from the mesh entity relationships. Everything that has been discussed in Section 3.3.3 can be easily automated except the renumbering of the pointer arrays. Application of indirections to local array accesses is straightforward once the indirection arrays have been obtained from utilities. Transferring loop limits from global to local ranges is possible due to organised renumbering from the utility, however every reference to the loop counter must be within the same indirection array. Consider the following code fragment

```
INTEGER   NUMBER_OF_GP_IN_ELEMENT(1:LOCAL_NUMBER_OF_ELEMENTS)
INTEGER   GP_IN_ELEMENT(1:MAX_NUM_GP_PER_ELE,1:LOCAL_NUMBER_OF_ELEMENTS)
INTEGER   PTR_ELE(1:NUMBER_OF_ELEMENTS)
INTEGER   PTR_GP(1:NUMBER_OF_GRID_POINTS)
```

```
REAL       XELE(1:LOCAL_NUMBER_OF_ELEMENTS)
REAL       YGP(1:LOCAL_NUMBER_OF_GRID_POINTS)

DO I = 1, NUMBER_OF_ELEMENTS
  IF ( OWNER_OF_ELEMENT(I) ) THEN
    IF ( I .LE. NTRI ) THEN
      NNODES = 3
    ELSE
      NNODES = 4
    END IF
    DO J = 1, NNODES
      XELE(PTR_ELE(I)) = XELE(PTR_ELE(I)) +
+                        YGP(PTR_GP((GP_IN_ELEMENT(J,PTR_ELE(I)))
    END DO
  END IF
END DO
```

Here the number of nodes for each element is determined by the global element number where the first `NTRI` elements are triangles. Since the reference to `I` in the conditional is not within the local number indirection then the loop limit cannot legally be altered. This is still a parallel loop operating on renumbered packed data but iterating globally. Such loops are easily identifiable by the tools along with the reason prohibiting transformation, in this case the reference to `I` in the conditional. In practice the users knowledge that `NTRI` represents the number of triangular elements can allow the user to provide the code to calculate a local value for `NTRI` and therefore enable the optimisation to continue. Other loops are independent of the decision for this loop and may still be able to transform the loop mask into loop limits.

The renumbering of pointer arrays to local numbering schemes is however a far more complicated problem. Consider the mesh storage examples from Section 6.2.2. The following code fragments are examples that may be used to access the structures along with codes to perform the renumbering.

a)

```
DO I = 1, LOCAL_NUMBER_OF_ELEMENTS
  DO J = 1, 4
    IF ( GP_IN_ELEMENT(J,I).NE.0 ) THEN
      XELE(I) = XELE(I) + YGP(PTR_GP(GP_IN_ELEMENT(J,I)))
    END IF
  END DO
```

```
END DO
```

The code to renumber GP_IN_ELEMENT is

```
DO I = 1, LOCAL_NUMBER_OF_ELEMENTS
  DO J = 1, 4
    IF ( GP_IN_ELEMENT(J,I).NE.0 ) THEN
      GP_IN_ELEMENT(J,I) = PTR_GP(GP_IN_ELEMENT(J,I))
    END IF
  END DO
END DO
```

This renumbering loop is a similar form to an inspector loop of the original code fragment.

b)

```
ICOUNT = 0
DO I = 1, LOCAL_NUMBER_OF_ELEMENTS
  ICOUNT = ICOUNT + 1
  DO J = 1, GP_IN_ELEMENT(ICOUNT)
    ICOUNT = ICOUNT + 1
    XELE(I) = XELE(I) + YGP(PTR_GP(GP_IN_ELEMENT(ICOUNT)))
  END DO
END DO
```

The code to renumber GP_IN_ELEMENT is

```
ICOUNT = 0
DO I = 1, LOCAL_NUMBER_OF_ELEMENTS
  ICOUNT = ICOUNT + 1
  DO J = 1, GP_IN_ELEMENT(ICOUNT)
    ICOUNT = ICOUNT + 1
    GP_IN_ELEMENT(ICOUNT) = PTR_GP(GP_IN_ELEMENT(ICOUNT))
  END DO
END DO
```

This renumbering loop is again a similar form to an inspector loop of the original code fragment.

c)

```
      ICOUNT = 0
      DO I = 1, LOCAL_NUMBER_OF_ELEMENTS
10      ICOUNT=ICOUNT+1
        XELE(I) = XELE(I) + YGP(PTR_GP(ABS(GP_IN_ELEMENT(ICOUNT))))
        IF (GP_IN_ELEMENT(ICOUNT).GT.0) GOTO 10
      END DO
```

The code to renumber GP_IN_ELEMENT is

```
      ICOUNT = 0
      DO I = 1, LOCAL_NUMBER_OF_ELEMENTS
10      ICOUNT=ICOUNT+1
        GP_IN_ELEMENT(ICOUNT) = PTR_GP(ABS(GP_IN_ELEMENT(ICOUNT))) *
     +                          SIGN(1,GP_IN_ELEMENT(ICOUNT))
        IF (GP_IN_ELEMENT(ICOUNT).GT.0) GOTO 10
      END DO
```

This renumbering loop is a similar form to an inspector loop of the original code fragment however the operation required to achieve renumbering with preservation of the sign to delimit element boundaries is not obvious. In effect, the multiplication by SIGN is the inverse of the ABS function used in the references to the array.

This is a difficult problem as the general case involves an extremely wide range of possibilities. The subtleties of detecting legal transformations guaranteeing that *all* of the array contents that are referred to in indirections are renumbered and also that those that are not in indirections (i.e. the number of nodes values in example (b)) are left alone, is extremely complex. This may lead to no renumbering of some pointer arrays in many cases, although, again, user involvement can ease this problem in some instances.

6.3 Summary

There exist many similarities between the methods used for automation of the parallelisation of structured and unstructured mesh codes. Some new additional techniques are required to extend the now established structured mesh methods to enable parallelisation of a wider range of codes using a vast range of data structures. Although not all optimisations can be applied in automatically in all cases, the code produced can closely resemble that produced manually as in the parallelisation of UIFS. The utilities developed for PUIFS have been simply adjusted to the general case.

Chapter 7

Other Parallel Issues

There comes a point when the research has to pause to allow the dissertation to be written. In the course of the work described in this dissertation a number of issues have surfaced that require some mention. An alternative title to this chapter may be Unfinished Business but while some of these issues will be addressed in the near future others remain the subjects of research that have yet to become accepted practice.

7.1 Are Further Improvements Possible?

The graphs given in Chapter 5 demonstrate how a range of results from poor to good with moderate parallelism can be transformed into good to excellent. It is fair to say that the poor results reflect poor communication performance especially in terms of the communication start up latency, this coupled with the good calculation performance of the parallel platform, leads to a poor calculation to communication ratio. Objective (v) requires scalability to massive parallelism. If this is to be achieved then excellent moderate scale parallelism is required. Given that a parallel machine is unlikely to ever return perfect performance all possible optimisations of the parallel code should be sought. It is apparent that there are a number of further improvements that could be implemented.

7.1.1 Layered Overlaps

The PUIFS implementation uses only two overlap structures. An element overlap and a grid point overlap. The size of each of these overlaps is determined by whether the flow code is used with or without the stress code. If the stress code is used then the size of each of the overlaps is increased to accommodate the extra dependence required. It is logical to allow for the definition of more than one overlap for each mesh entity. This will provide a small improvement in efficiency within the flow portion of a run that also involves stress. The reduction in run times will be more apparent for bandwidth limited problems. Implementation of layered overlaps within PUIFS is a reasonably simple optimisation that could be implemented with a duplication of the communication schedules. For automated parallelisation the outcome of inspector loops described in Section 6.2.3 is a set of dependencies which can result in many layers for each overlap.

7.1.2 Machine Topology Profile

In spite of what parallel machine manufacturers may claim there will always be a distance related communication cost. This cost becomes more significant as the number of processors increases. To quantify the variations in latency and bandwidth a code has been developed which measures the communication performance of a parallel machine. Latency is measured by the simple method of sending a short message between each processor on the parallel machine. Similarly bandwidth is measured by sending a large message between each processor. These measurements are initially carried out with only one message being passed at any one time, and then with every processor communicating simultaneously. This provides a peak and a saturated performance measure that may be expressed as a weighted graph (matrix) that describes the communication performance between each pair of processors. What is immediately apparent is the non-uniform performance described by the graph. Such a weighted graph can be obtained quickly, at run time, and then used by the partitioning code to ensure that the mesh partition produced is appropriate for the measured machine communication profile as opposed to a notional topology that may not be reflected in actual communication performance. It has been

demonstrated in Chapter 5 that reflecting the processor topology in the mesh partition provides a performance improvement. It is therefore anticipated that this scheme will provide improved performance across a range of parallel machines without the need to understand or specify the architecture of the machine.

7.1.3 Dynamic Load Balance

The test cases used in this thesis have been partitioned to achieve a static load balance. This has been achieved through balancing the number of elements in each partition. For a constant element shape this can give a good load balance. There are many reasons why good load balance may not be achieved or maintained.

The variation in computational load for different element types (shapes) can be accommodated to some extent in the partitioning process as discussed in Section 3.2.4, however the exact computational balance can only be determined at run time. Changing physics in an application can affect the amount of computation per element or grid point. For example phase change (solidification) can lead to more complex physical processes requiring extra computation, e.g. latent heat release may effect some elements and not others.

The parallel machine may not have homogeneous performance, that is some processing elements may be faster or slower than others. This is especially true for workstation networks. Again such a variation can be accommodated to an extent in the partitioning process but cannot be accurately predicted. It is often the case for workstation networks that the machines may be used by other jobs, again causing an imbalance in processing performance.

A dynamic load balancing scheme is required to re-distribute the work over the processors so as to minimise idle time. Many of the recently developed partitioning schemes address such repartioning as a parallel task. There exist however some questions that require investigation.

How often should the balancing process be carried out? Re-partitioning to redress load imbalance is an overhead that requires optimisation between the degree to which

load imbalance may be tolerated and the cost of re-partitioning.

How much of the load should be moved? If the load imbalance is caused by movement of the computational load or changes in the computational resource then the best that can be hoped for is to use some expression to anticipate the movement of load that will redress the imbalance. Where changes in the computational resource is caused by outside agencies such as the submission of other jobs to a network it is impossible to predict the future partition requirements. Given that a parallel code executes only as fast as its slowest processor this can cause difficult problems for such open systems.

How to avoid cycling? If the parameters used to redistribute the load are inappropriate then the load balancing scheme may cycle the load between processors or even circulate the problem around the parallel machine. Methods similar to taboo search can be used to reduce such incidence but cycle recognition remains unclear.

Undoubtedly dynamic load balancing is not only desirable but may be necessary for some parallel applications. However some of the open problems have no easy solution. The scope of the problem is reasonably clear but an elegant solution has yet to be identified.

7.1.4 Other Communication Schemes

With a small mesh size latency becomes the overridingly dominant communication cost. Some success has been demonstrated with communication schemes that help to reduce the latency cost. Rather than considering the machine to be connected as a mesh, consider the machine to be connected as a star. All communication is transacted via the processor at the hub [GWZ95]. This has reduced the number of processor interconnections to P. For example an overlap update for a processor (sub-domain) inside a two dimensional processor array would be likely to incur up to eight latencies. A processor in a hub connected array would incur only one latency. Partitioning would be much the same as described in this thesis, with a low cut edge count partition likely to give the best performance. Only the communication would alter, all overlap data would be exchanged via the processor at the hub. Each processor can pass data required by other

processors in one packet to the central processor. The hub processor having accumulated data from all other processors sends back to each processor the data corresponding to its overlap. Of course the scheme would require some degree of asynchronous communication to avoid a bottleneck at the hub processor. Also the hub processor would not be able to carry out a full share of the workload and so anticipating a static load balance would be a problem. Ultimately the scheme would not scale far as the communication load on the central processor becomes too great. However the premise for the scheme was to alleviate latency bound problems, i.e. small mesh sizes, which would not be expected to scale far anyway. So if the requirement was to improve performance for small problems this could be a worthwhile investment of effort. Workstation networks with PVM for example are a suitable platform for this strategy as such a system incurs a very high latency and supports non blocking communications.

7.2 Difficult Problems

7.2.1 Inhomogeneous Problems

Figure 7.1 shows a simple foil mesh partitioned into four sub-domains, each containing the same number of elements. This has achieved a balance of elements across each processor but it is necessary to balance the load across all solvers. In this example only flow is solved for in the space around the foil and only stress is solve within the foil. To achieve a load balance the nature of these physical domains must be incorporated into the partitioning scheme. A more balanced partition may appear more like that shown in Figure 7.2. Here the balance of elements across processors has been maintained within the foil and outside the foil but at the cost of an increased and imbalanced communication.

A dynamic load balancing scheme is in this case required to acknowledge the differing physical domains. Code execution time *within each solver* could possibly be used to direct the redistribution of a mesh in accordance with the physical domains. Solidification problems, for example, present severe difficulties to this type of scheme due to the massive migration of elements required as the computational load moves from possibly the entire

determine if any contact between otherwise unconnected parts of the mesh has occurred. This involves a dependence between all parts of the mesh. Some work has been done to restrict possible dependence to localised parts of a mesh [GMD95a]. Similar difficulties arise for moving mesh problems and particle based codes such as molecular dynamics.

7.3 Are there any alternatives?

The strategies discussed in this thesis follow the now accepted path of development that attempts to reproduce a serial code as faithfully as possible on a DM-MIMD machine. There are however some developments that may alter the way in which parallel machines are used.

7.3.1 Parallel Mesh Generation

An alternative solution to the problems presented by a very large mesh in relation to the capacity of a single processor is parallel mesh generation. This scheme begins with a geometric decomposition of the problem space into P sub-domains each of which are then meshed in parallel [CJL+89, HJ94]. It is arguable that the mesh quality is hard to control. But the matter of mesh quality is a difficult issue which is beyond the scope of this thesis. Also relevant is the inability to compare serial with parallel when there is no serial case. The measure of the success of a parallel exercise remains a comparison of the parallel results with serial results. This viewpoint may have to change with the arrival of highly parallel processing where $P > 1024$. Such a machine would invite the application of meshes of such magnitude that a comparison with serial performance would be out of the question and even visualisation of the full mesh may be impractical. Analysis of the mesh quality would in that case have to be automatic and so any measure that may be applied to mesh quality may also be used to improve the mesh to the point of conformance with stipulated criteria of quality (Zen and the art of mesh generation).

7.3.2 Parallel Visualisation

The final product of computational mechanics is normally some form of graphical image. As computational techniques have become more sophisticated so too have the techniques of visualisation. It is now commonplace to transform the results of scientific computation into animated three dimensional images. Such animations may be supplemented with advanced techniques such as particle tracking and stream lines/ribbons. The process of transforming data sets into graphical presentations is a computationally demanding exercise that lends itself to parallel processing. There is an attractive logic in removing the step of writing data sets from parallel computation to file and producing images instead [Hei94]. The generation of images is however usually an interactive process as view angles, lighting, perspectives, etc are manipulated to produce the required result. As such the visualisation may become a repetetive labour intensive task, not a task well suited to batch processing. Nevertheless one of the aims of parallel processing is to reduce the run time of programs to the point at which interactive computational mechanics becomes a possibility. High speed run time visualisation would be a great asset to such an undertaking.

7.3.3 Virtual Shared Memory

A shared memory parallel machine is without doubt a far easier machine to program than a distributed memory machine. With a shared memory machine a problem can exist in machine memory in the same form as the serial without the need for renumbering or overlaps and overlap updating. All that is required for such parallel processing is the determine an appropriate partition for compute masking. Dynamic load balancing for example becomes far simpler as only the compute masks require adjustment to redress balance. The advantages for adaptive meshing and inhomogeneous problems are manifold. There remains however the problem of scalability. The memory bandwidth in a shared memory machine does not scale with the number of processors. For this reason current shared memory machines are normally less than 16 processors and for most CM codes SM machines operate best at less than 8 processors. Virtual Shared Memory

(VSM) machines are systems that allow the entire machine memory to appear as shared memory even though it may not be actually shared. A VSM machine may be a distributed memory machine with a software harness that allows the processors to address the memory on other processors. A VSM machine may be a cluster of shared memory machines, again with a software harness that allows the memory to appear shared amongst all processors. Some manufacturers have employed elaborate hardware, memory and cache arrangements to provide workable VSM with distributed memory. Many manufacturers and academics claim that the future of parallel processing lies with VSM. Indeed the advantages that such systems offer for simplicity of programming are clear. But unless memory bandwidth scales with the number of processors then scalability will not be achieved. The advocated VSM model is one in which compilers and operating systems communicate as often as necessary. However the cost of ignoring processor topology on a DM machine, even with a well partitioned mesh is clearly demonstrated in Chapter 5. If in addition all concept of cut edge were ignored the communication requirements of a CM code will become astronomical. Since we can organise unstructured meshes onto processor topologies with significant performance gains and little human effort (Chapter 6) there appears to be little to be gained and a lot to be lost through VSM. Perhaps the more enlightened view is one recently voiced at a conference; If manufactureers allow VSM programming on their hardware then automatic compilation of existing code onto small numbers of processors will encourage use (purchase) of parallel machines. Having persuaded people to use (buy) the machine they may then be encouraged to optimise their application through the use of message passing techniques. Cost effectiveness remains an overridingly important criteria in the commercial success of a system. An architecture that is gaining popularity is to produce highly cost effective 4 – 8 processor SM systems that may be interconnected with low latency, high bandwidth interfaces. This makes such systems highly attractive as they may be used very effectively for running multiple low P jobs or less effectively using VSM to allow very large problems to be accommodated with little or no parallel skills being required. Optimising code for such a platform will nevertheless benefit from the techniques described in

this dissertation to avoid the inevitable bottlenecks as data moves between the shared memories.

Chapter 8

Conclusions

8.1 Were the Objectives Met?

A number of objectives for parallelisation of an unstructured mesh CM code were set out in the introduction to this thesis. To what extent have these objectives been achieved?

8.1.1 Objective (i) Minimise the Changes to the Original Algorithm

The entire test case program UIFS has been parallelised with only one simple algorithmic change being required. The Jacobi and Conjugate Gradient algorithms are both reproduced identically in parallel. Rounding errors are however subject to coefficient evaluation order and can therefore give rise to variations in the numerical values produced by the code despite there being no actual algorithmic change. In the interest of meeting objective (iii) (and to some extent (ii)) the Gauss Seidel SOR algorithm has been modified in the parallel scheme proposed in this thesis. While the results produced by the parallel GS-SOR solver are numerically dependent on factors such as the number of processors and the mesh partition, these changes are qualitatively insignificant. The results produced by the serial code are qualitatively reproduced by the parallel code. In practice the variation between serial and parallel GS-SOR results caused by algorithmic modification are no greater than the variation between the serial and parallel CGM results which do not arise from algorithmic modification. It should be possible however to

produce pathological test cases that are extremely sensitive to numeric accuracy. But in such a case the results produced by the serial code are highly questionable. This raises an issue in that if the parallel results differ significantly from the serial then we should suspect that the serial results are of questionable validity. It is clear that objective (i) has been achieved for the three solvers covered in this thesis. There exist however a great many other solvers in use in CM codes. It cannot be guaranteed that all solvers will be so amenable to parallelisation. Parallelism probably exists in all algorithms but the limitation of achievable parallel machine performance (latency and bandwidth) restricts the feasibility of some parallel solutions. There are some classic examples from the structured mesh CM codes of solvers that are either difficult or impractical to parallelise. Some of these solvers have been elevated to the status of benchmarks in the NASPAR suite [BBLS93]. The ADI solver and LU factorisation (APPLU) are two examples that have achieved some notoriety in their difficulty for successful parallelisation. A lesson that is now being accepted by the CM community is that if the solver is unsuited to parallelisation then it should be replaced by one of the many solvers that are highly paralleliseable. The replacement solver may not give such good performance on one processor but the parallel performance can usually justify the substitution. This has been the case with the parallelisation of the highly successful PHOENICS structured mesh CFD code from CHAM [CHA94]. Like UIFS this code offers a range of solvers, in particular the highly efficient conjugate gradient with ILU preconditioning is used. This solver has caused a number of difficulties for parallelisation and so the solution adopted for the parallel code was to use the slightly less efficient but highly parallel Jacobi preconditioned conjugate gradient solver [GMD95b]. The strategy adopted by CHAM for the Gauss-Seidel solver is the same as used in this thesis.

8.1.2 Objective (ii) Minimise the Visibility of the Parallel Code

Whilst the parallel code can hardly be described as invisible, the layered library strategy described in Section 2.4 has allowed the bulk of the parallel code to be hidden. The concept of the PUTILS library is to provide a barrier that obscures the parallel imple-

mentation and the parallel machine from the view of a code author. This is achieved by providing a source code perspective that requires only a minimal knowledge of parallelism to understand and use the library routines. For the PUIFS code this exercise has proved to be a great success, as many of the routines have required no modification whatsoever. Of the 209 subroutines in UIFS only 71 have required some alteration to function in parallel. Of these 71 routines that require either calls to the PUTILS routines or use of the data in `puifs.inc` the intrusion into the original routines has been in most cases at an acceptably low level (c.f. Appendix C). Unfortunately 35 of the 71 parallelised routines are i/o routines that have required significant modification. Little can be done to ameliorate this problem especially while parallel i/o remains an uncommon hardware feature. In actuality the i/o problem is too great as the i/o routines are (despite their size) some of the simpler routines in the code and so re-authoring them for parallel functioning is not a great task, especially with the PUTILS routines available.

From the user perspective concealment of parallelism has shown great success. No modifications of the problem definition and specification are required for the problem to be run on a parallel machine. Partitioning and decomposition of the problem has been implemented in PUIFS as a transparent run-time process. Even the restart files may be used to move sequential runs between serial and parallel machines and between differing numbers of processors on parallel machines. The small problem of binary file compatibility can occur when moving between systems but is no more complicated in the parallel case as in a transfer between differing serial machines. No additional input is required from the user other than the number of processors that are to be used. This single integer is not however trivial information. It is conceivable that the choice of the number of processors to use could be made by the program. For a very small problem it can be difficult to obtain speed-up on a parallel machine with a poor calculation to communication ratio. Having run the parallel code with a range of problem sizes a profile of the machines returned performance is obtained. Such a profile can be used to determine the maximum number of processors that will return an 'effective' speed-up for the size of problem to be run. The definition of effective need not be static but could

be influenced by the demand history placed on the machine. This opens up numerous possibilities for job scheduling to maximise return from parallel resources.

8.1.3 Objective (iii) Maximise Parallel Efficiency

Chapter 5 presents some results that range from very poor (Figure 5.15) to very good (Figure 5.35). Clearly the returned performance is highly problem dependent. For example a guaranteed slow-down is possible given a small enough test case. Likewise near perfect performance could be obtained with a constant problem size per processor of near the maximum that can be accommodated. In the case of PUIFS on the Transtech Paramid at the University of Greenwich where the smallest nodes have 16MBytes of memory with a triangular mesh then approximately 20,000 elements can be accommodated per processor or 560,000 elements over the entire machine. This problem would occupy approximately 500MBytes of memory Extrapolation of the speed-up curves gives an estimated speed-up of over 24 or approximately 90% efficiency. 16 of the Paramid nodes have 32MBytes of memory which would allow a problem of more like 30,000 elements per processor or 480,000 elements in total. Extrapolation gives a speed-up of around 15.5 and efficiency at around 95%. As discussed in Chapter 5 the efficiency of a parallel code is a highly machine dependent measurement. It would seem from the Paramid results that the machine dependency is most noticeable for smaller sized problems, or more accurately for smaller meshed problems. There exist many real UIFS problems with mesh sizes well below 3,000 elements for which the run times on a workstation are about a week. PUIFS on the Paramid can only be of limited help with such problems. With a small mesh PUIFS problem the Paramid machine is latency bound and the returned performance consequently poor.

Topology mapping the mesh partition to the Paramid has been demonstrated to provide improved performance. This is especially noticeable at the point in the speed-up curve where performance falls off. Clearly the commonly accepted criteria of mimising the number of cut edges in the partition does not necessarily provide the best performance. Several avenues of development that provided a highly significant improvement

in performance were discussed in Section 5.4. The extent to which such improvements can be made is again platform dependent. However all possible improvements must be explored and implemented if further gains can be made in performance. The modifications required to implement the optimisations were contrary to objective (ii) in that significant alteration of the source code was required. However only the three solver routines are affected by these modifications which is a small disadvantage in comparison to the enormous performance improvement.

8.1.4 Objective (iv) Portability to Most DM MIMD Platforms

It is now widely accepted that use of one of the well known message passing libraries with Fortran77 code provides a highly portable parallel code. Use of the CAPTools libraries has provided an improved portability interface than direct use of a message passing library. The majority of the currently used message passing interfaces are supported by the CAPTools libraries. Not only does this extend the portability to all of the message passing interfaces that are supported by `CAPLIB` but it allows the choice of the most efficient supported library to be made. Porting of the libraries to other interfaces is not automatic but has at least been reduced to an easily manageable task, the porting of `CAPLOW`.

8.1.5 Objective (v) Scalability of Computation

Computational scalability is another highly machine dependent parameter. Computational scalability has been shown to be achievable *provided* that the problem size is large enough. This does not however fully address the issue. The returned performance presented in Chapter 5 quite clearly does not scale well for small problems and especially not for the stress code. Why not? Latency is the limit on scalability of computation not only for the Transtech Paramid but also for a great many other parallel systems.

8.1.6 Objective (vi) Scalability of Memory

Scalability of memory has clearly been achieved up to the point at which a description of the mesh or a globally sized vector variable may be accommodated within the memory of one processor. This takes scalability of memory way beyond the limit of the 28 processors available using the University of Greenwich Paramid. How far? For the PUIFS code the current strategy can take scalability to around 60 processors. After that more effort will be required to fully parallelise some of the i/o operations.

8.1.7 Objective (vii) Automate the Parallelisation Process

Automation of the entire process of parallelisation will eventually be realised. The CAPTools project has acknowledged that this process can be enhanced with user supplied knowledge and so has provided an interactive toolkit to automate as far as possible the process of parallelisation of mesh based codes. The dependence analysis already available in CAPTools provides a powerful analysis of unstructured mesh codes. This thesis has developed strategies for parallelisation of unstructured mesh codes based upon those developed for structured mesh codes. The CAPTools libraries have been used to develop utilities for unstructured meshes that point the way for generic techniques that may be used in an automated parallelisation process. The techniques developed in this thesis will eventually be incorporated into the CAPTools package to extend the scope of parallelisation to irregular mesh based codes. Some open problems discussed in Chapter 6 remain but do not obstruct the development of CAPTools towards unstructured mesh codes.

8.2 Summary

Why did we trouble ourselves with parallelisation in the first place? Nobody really wants parallel processing, what is really required is a larger, faster serial processor. That way we do not have to expend any effort parallelising codes and our time can be used more profitably elsewhere. But parallel processing is inevitable. There are a number of reasons

that keep bringing us back to parallel concept. However large and fast a serial processor can be built there will always be the temptation to connect several of them together to create a single system with greater power. However large a problem we are currently solving we want to be able to solve a larger problem. A more pragmatic reason is simply the economics of producing a supercomputer. Economy of scale in the development of workstation technology makes a highly parallel machine based on this technology the most cost effective approach to high performance computing. The arguments concerning the optimal architecture to adopt for such a parallel machine will probably continue for some time but the common ground remains the same: Connect together a large number of state of the art processors and memory to produce a single high performance system. It is therefore essential to develop the skills required to use such a system efficiently.

Appendix A

Parallel Utilities

A.1 Parallel Included Declarations

The following is the include file `puifs.inc` that declares the extra variables required for parallel processing. These declarations are scalable to the extent that a buffer is required on the i/o processor that can hold a globally sized set of coordinates or a globally sized variable in order to reconstruct the coordinates or a variable prior to writing to file. In this instance `MAXBUF` must be declared as the greater of either `MAXELE` or twice the size of `MAXGPT`. One difficulty with included declarations in F77 is the explicit length declaration of array sizes. Any change to the declaration of `MAXBUF` requires recompilation of all sources that include `puifs.inc`.

```
C=================================================================
C     Parallel UIFS include file
C     K. McManus  12th June 1993
C     University of Greenwich
C     London UK
C=================================================================
C
C     MAXHLO   Maximum halo size
C     MAXBUF   Maximum size of buffer
C     XTOTEL   Total number of elements including halos
C     XTOTGP   Total number of grid points including haloes
C     XNETYP   Total number of element types including haloes
C     GTOTEL   Global total number of elements
C     GTOTGP   Global total number of grid points
C     BUFLEN   Buffer length - scratch
```

```
C     PROCNUM  This processors number
C     NPROC    Number of processors (sub-domains)
C     MASTER   Logical true if processor number one
C     ELINDX   Element index, global element numbers for this subdomain
C     PTINDX   Point index, global point numbers for this subdomain
C     HEINDX   Halo element communication index
C     HPINDX   Halo point communication index
C     BUFFER   Integer buffer - big
C     IUFFER   Integer buffer
C     RBUFER   Floating point buffer - big
C     FBUFER   Floating point buffer
C     DBUFER   Double precision buffer
C
C=============================================================

      INTEGER      MAXHLO, MAXBUF
c  note MAXBUF must be divisible by 16 for data alignment
      PARAMETER    ( MAXBUF = 160000 )
      PARAMETER    ( MAXHLO = MAXBUF/40 )
      INTEGER      XTOTEL, XTOTGP, XNETYP
      INTEGER      GTOTEL, GTOTGP, BUFLEN
      INTEGER      PROCNUM, NPROC
      LOGICAL      MASTER

      INTEGER      ELINDX(0:MAXBUF/2)
      INTEGER      PTINDX(0:MAXBUF/2)
      INTEGER      HEINDX(1:MAXHLO)
      INTEGER      HPINDX(1:MAXHLO)

      INTEGER      BUFFER(1:MAXBUF)
      INTEGER      IBUFER(1:MAXBUF)
      REAL         RBUFER(1:MAXBUF)
      REAL         FBUFER(1:MAXBUF)
      REAL*8       DBUFER(1:MAXBUF)


      INTEGER      ELEMENT, NODE, D_NODE
      PARAMETER    ( ELEMENT = 1 )
      PARAMETER    ( NODE    = 2 )
      PARAMETER    ( D_NODE  = 4 )

c the buffers are arranged into overlapping memory space
      EQUIVALENCE  ( DBUFER(1), RBUFER )
      EQUIVALENCE  ( DBUFER(MAXBUF/2 + 1), FBUFER )
      EQUIVALENCE  ( BUFFER, RBUFER )
      EQUIVALENCE  ( IBUFER, FBUFER )

      COMMON     /PCOMM/ XTOTEL, XTOTGP, XNETYP, GTOTEL, GTOTGP,
     @                   PROCNUM, NPROC, MASTER,
```

```
@                   ELINDX, PTINDX, HEINDX, HPINDX,
@                   BUFLEN, DBUFER,
@                   ELEMENT, NODE, D_NODE
 SAVE       /PCOMM/
```

A.2 Parallel Utility Library

The routines provided by the parallel utility library discussed in Section 2.4.1 are as follows:

- `INITIALISE()`

  Sets up the parallel configuration. This initialises the variables `NPROC` and `PROCNUM` which remain hidden in the common data `PCOMM`.

- `HALT()`

  Shuts down parallel processing.

- `CHECK()`

  Checks to see if the processors are responding. Used only to provide confidence check.

- `BCAST( VARIABLE, VARIABLE_LENGTH )`

  Broadcasts a `VARIABLE` of size `VARIABLE_LENGTH` from the master processor to all processors. All processors are left with an identical value for `VARIABLE`. This could have been implemented with a processor number as an argument to indicate which processor is broadcasting. This was not however found to be required, mainly as a consequence of the SPMD paradigm.

- `GSUMR( REAL_VARIABLE )`

  Returns the global sum of the `REAL_VARIABLE` to all processors.

- `GMAXR( REAL_VARIABLE )`

  Returns the global maximum of the `REAL_VARIABLE` to all processors.

- `GSUMD( DOUBLE_VARIABLE )`

  Returns the global sum of the `DOUBLE_VARIABLE` to all processors.

- `GMAXR( DOUBLE_VARIABLE )`

  Returns the global maximum of the `DOUBLE_VARIABLE` to all processors.

- `GOR( BOOLEAN_VARIABLE )`

  Returns the global OR of the `BOOLEAN_VARIABLE` to all processors.

- `GAND( BOOLEAN_VARIABLE )`

  Returns the global AND of the `BOOLEAN_VARIABLE` to all processors.

- `SWAP( VARIABLE, SPATIAL_REFERENCE )`

  Exchanges the `VARIABLE` values in the overlaps to give consistent data across all processors. `SPATIAL_REFERENCE` may be any of `ELEMENT`, `NODE` or `D_NODE`.

- `SCATTER( VARIABLE, SPATIAL_REFERENCE )`

  Distributes a global `SPATIAL_REFERENCE` based variable from the master processor to be a local variable on all processors. Used only in i/o routines.

- `GATHER( VARIABLE, SPATIAL_REFERENCE )`

  Rebuilds a global `SPATIAL_REFERENCE` based variable onto the master processor from local variables on all processors. Used only in i/o routines.

- `TOPROC( PROCESSOR_NUMBER, BUFFER, VARIABLE, VARIABLE_LENGTH )`

  Sends `BUFFER` of length `VARIABLE_LENGTH` from the master processor into `VARIABLE` on `PROCESSOR_NUMBER`. This call requires a `PROCESSOR_NUMBER` and is used only in i/o routines.

- `FROMPROC( PROCESSOR_NUMBER, BUFFER, VARIABLE, VARIABLE_LENGTH )`

  Sends `VARIABLE` of length `VARIABLE_LENGTH` from `PROCESSOR_NUMBER` into `BUFFER` on the master processor. This call requires a `PROCESSOR_NUMBER` and is used only in i/o routines.

- `ASWAP( VARIABLE, SPATIAL_REFERENCE, SWAP_ID )`

  Performs an asynchronous (non-blocking) exchange of the `VARIABLE` values in the overlaps to give consistent data across all processors. `SWAP_ID` is a unique identifier for the communication.

- `SYNC( SWAP_ID )`

  Waits until the message identified by `SWAP_ID` is complete.

Appendix B

Partition List

The partition listed here corresponds to the partitioned mesh shown in Figure 4.7

The first entry is the number of elements N (nodes in graph), the second is the number of partitions P. There then follows a list of N numbers giving the partition to which the element belongs.

```
42
3
2   3   2   1   3
1   1   1   1   2
3   2   1   1   3
1   1   3   2   3
2   1   3   2   3
2   3   2   3   1
3   3   3   2   2
3   1   2   2   2
1   1
```

Appendix C

Parallel Iterative Solvers

The code listed here is the original serial code that has been modified to function in parallel by the addition of the code highlighted in bold. The additional subroutines called can be found in the PUIFS utility library and the additional variables used are from putils.inc, both are listed in Appendix A.

These routines have met the requirements of objectives (i), (ii), (iii) and (iv);

(i) The algorithms for Jacobi and DPCGM are unchanged, and for Gauss SOR minimally changed.

(ii) The changes made to the serial code are minimal and hopefully comprehendable without extensive knowledge of parallel processing.

(iii)Given a well balanced partition the parallel efficiency is potentially high as little communication is required.

(iv) Portability is achieved through the use of library functions.

Whether scalability (requirement (v)) has been achieved is dependent on the implementation of the global commutative functions.

C.1 Jacobi Solver

```
C-----------------------------------------------------------------------
C  Subroutine  JACOBI  (JACOBI) scheme
C
C
C  Author       : P.Chow  23rd March 1989
C                 K. McManus 23rd September 1993
C                 Centre for Numerical Modelling & Process Analysis
C                 University of Greenwich, London, England.
C
C  Description : Solve Ax = b using Jacobi iterative scheme.
C
C  Variables   :
C IN   RMETHD  -  Residual method.
C IN   TOLVAL  -  Tolerance value.
C IN   MAXITR  -  Maximum number of iteration.
C IN   TOMITR  -  To maximum iteration.
C IN   TOTELP  -  Total number of grid points per element.
C IN   TOTELE  -  Total number of element.
C IN   SYSINX  -  System matrix index.
C IN   SYSMAT  -  System matrix A.
C IN   B       -  B vector.
C I&O  X       -  Solving variable X.
C OUT  RESVAL  -  Residual values.
C OUT  NITERS  -  Number of iteration taken.
C OUT  BIGRES  -  Biggest residual value.
C OUT  CONVER  -  Convergent indicator.
C WSP  OX      -  Old X value (work space).
C
C-----------------------------------------------------------------------
      SUBROUTINE JACOBI ( RMETHD, TOLVAL, MAXITR, TOMITR, TOTELP,
     @                    TOTELE, SYSINX, SYSMAT, B     , X     ,
     @                    RESVAL, NITERS, BIGRES, CONVER, OX     )

        INTEGER     RMETHD, MAXITR, TOTELP, TOTELE, NITERS
        INTEGER     SYSINX(0:TOTELP,1:TOTELE)
        REAL        TOLVAL, BIGRES
        REAL        SYSMAT(1:TOTELP,1:TOTELE)
        REAL        B     (1:TOTELE), X     (1:TOTELE)
        REAL        RESVAL(1:TOTELE), OX    (1:TOTELE)
        LOGICAL     TOMITR, CONVER

C  Commons
       INCLUDE 'puifs.inc'

C  Local Constants
        INTEGER     HEADER, IZERO , IONE
```

```
      PARAMETER   ( HEADER = 0, IZERO  = 0, IONE   = 1 )

C  Local Variables
      INTEGER     ISTART, IEND  , ISTEP , I     , J
      LOGICAL     DONE

      NITERS = IZERO

100   CONTINUE
         NITERS = NITERS + IONE

C          DO 150 I = IONE, TOTELE - operate locally on the overlap
         DO 150 I = IONE, XTOTEL
            OX(I) = X(I)
150      CONTINUE

         DO 300 I = IONE, TOTELE
            X(I) = B(I)
            DO 200 J = IONE, SYSINX(HEADER,I)
               X(I) = X(I) + SYSMAT(J,I) * OX(SYSINX(J,I))
200         CONTINUE

            RESVAL(I) = ABS ( X(I) - OX(I) )
300      CONTINUE

         CALL ERESID ( RMETHD, TOTELE, RESVAL, X     , BIGRES )

         CONVER = BIGRES .LE. TOLVAL
         DONE   = (NITERS .GE. MAXITR) .OR.
     @            (CONVER .AND. (.NOT. TOMITR))

         CALL SWAP ( X, 'E' )

      IF ( .NOT. DONE ) GOTO 100

      RETURN
      END
```

The solver calls ERESID to evaluate the residuals at each iteration. This routine in turn calls LNORMS to evaluate the residual norms. ERESID is unchanged in parallel, but is listed here with LNORMS to show the call to LNORMS which requires a global commutative operation.

```
C-----------------------------------------------------------------
C  Subroutine  ERESID  (E)rror (RESID)ual
C
```

```
C  Author       : P.Chow  1st July 1992
C                 Centre for Numerical Analysis & Process Control
C                 University of Greenwich, London, England.
C
C  Description : Evaluate the residual value given residual and
C                 current values.
C
C  Variables    :
C IN   METHOD  -  Method of residual evaluation.
C                 1  -  Absolute L-1 norm.
C                 2  -  Absolute L-2 norm.
C                 3  -  Absolute L-Infinity norm.
C                 4  -  Relative L-1 norm.
C                 5  -  Relative L-2 norm.
C                 6  -  Relative L-Infinity norm.
C
C IN   TOTELE  -  Total number of elements.
C IN   RESDIF  -  Residual difference.
C IN   CURVAL  -  Current value
C OUT  RESVAL  -  Residual value.
C
C-----------------------------------------------------------------------
      SUBROUTINE ERESID ( METHOD, TOTELE, RESDIF, CURVAL, RESVAL )

      INTEGER     METHOD, TOTELE
      REAL        RESVAL
      REAL        RESDIF(1:TOTELE), CURVAL(1:TOTELE)

C  Local Constants
      INTEGER     ITHREE
      PARAMETER   ( ITHREE = 3 )

C  External User Defined Functions
      REAL        LNORMS, TOZERO
      LOGICAL     EQZERO
      EXTERNAL    LNORMS, TOZERO, EQZERO

C  Local Variables
      INTEGER     SLNORM, I
      REAL        ZETA

      SLNORM = METHOD
      IF ( METHOD .GT. ITHREE ) THEN
         SLNORM = METHOD - ITHREE

         ZETA = TOZERO ()

         DO 100 I = 1, TOTELE
            RESDIF(I) = RESDIF(I) / ( ABS ( CURVAL(I) ) + ZETA )
```

```
100       CONTINUE
       END IF

       RESVAL = LNORMS ( SLNORM, TOTELE, RESDIF )

       RETURN
       END

C-----------------------------------------------------------------------
C  Real Function  LNORMS  (L)-(NORMS)
C
C  Author       : P.Chow  1st July 1992
C                 K. McManus 23rd September 1993
C                 Centre for Numerical Analysis & Process Control
C                 University of Greenwich, London, England.
C
C  Description : Evaluate the L-1, L-2 or L-infinity norm of a vector
C
C  Variables    :
C IN   METHOD  -  Method of norm.
C                 1  -  L-1 norm.
C                 2  -  L-2 norm.
C                 3  -  L-infinity norm.
C IN   TOTELE  -  Total number of elements.
C IN   VECTOR  -  Vector of real numbers.
C
C-----------------------------------------------------------------------
       REAL FUNCTION LNORMS ( METHOD, TOTELE, VECTOR )

       INTEGER      METHOD, TOTELE
       REAL         VECTOR(1:TOTELE)

C  Local Constants
       INTEGER      L1NORM, L2NORM, LINORM, IONE
       REAL         ZERO
       PARAMETER    ( L1NORM = 1, L2NORM = 2, LINORM = 3, IONE    = 1 )
       PARAMETER    ( ZERO   = 0.0 )

C  Local Variables
       INTEGER      I
       REAL         RESVAL

       RESVAL = ZERO

       IF ( METHOD .EQ. L1NORM ) THEN
          DO 100 I = IONE, TOTELE
             RESVAL = RESVAL + VECTOR(I)
100       CONTINUE
          CALL GSUMR ( RESVAL )
```

```
      ELSE IF ( METHOD .EQ. L2NORM ) THEN
         DO 200 I = IONE, TOTELE
            RESVAL = RESVAL + VECTOR(I) * VECTOR(I)
200      CONTINUE
         CALL GSUMR ( RESVAL )
         RESVAL = SQRT ( RESVAL )
      ELSE IF ( METHOD .EQ. LINORM ) THEN
         DO 300 I = IONE, TOTELE
            IF ( VECTOR(I) .GT. RESVAL ) RESVAL = VECTOR(I)
300      CONTINUE
         CALL GMAXR ( RESVAL )
      END IF

      LNORMS = RESVAL

      RETURN
      END
```

C.2 Gauss-Seidel Solver

```
C-----------------------------------------------------------------------
C  Subroutine  SORSCH  (SOR) (SCH)eme
C
C
C  Author       : P.Chow  23rd March 1989
C                 K. McManus 23rd September 1993
C                 Centre for Numerical Modelling & Process Analysis
C                 University of Greenwich, London, England.
C
C  Description : Solve Ax = b using SOR iterative scheme.
C
C  Variables   :
C IN   RMETHD  -  Residual method.
C IN   RELAXA  -  Relaxation value.
C IN   TOLVAL  -  Tolerance value.
C IN   MAXITR  -  Maximum number of iteration.
C IN   TOMITR  -  To maximum iteration.
C IN   TOTELP  -  Total number of grid points per element.
C IN   TOTELE  -  Total number of element.
C IN   SYSINX  -  System matrix index.
C IN   SYSMAT  -  System matrix A.
C IN   B       -  B vector.
C I&O  X       -  Solving variable X.
C OUT  RESVAL  -  Residual values.
C OUT  NITERS  -  Number of iteration taken.
C OUT  BIGRES  -  Biggest residual value.
C OUT  CONVER  -  Convergent indicater.
C
C-----------------------------------------------------------------------
      SUBROUTINE SORSCH ( RMETHD, RELAXA, TOLVAL, MAXITR, TOMITR,
     @                    TOTELP, TOTELE, SYSINX, SYSMAT, B     ,
     @                    X     , RESVAL, NITERS, BIGRES, CONVER )

      INTEGER     RMETHD, MAXITR, TOTELP, TOTELE, NITERS
      INTEGER     SYSINX(0:TOTELP,1:TOTELE)
      REAL        RELAXA, TOLVAL, BIGRES
      REAL        SYSMAT(1:TOTELP,1:TOTELE)
      REAL        B     (1:TOTELE), X     (1:TOTELE)
      REAL        RESVAL(1:TOTELE)
      LOGICAL     TOMITR, CONVER

C  Local Constants
      INTEGER     HEADER, IZERO , IONE
      PARAMETER   ( HEADER = 0, IZERO  = 0, IONE   = 1 )

C  Local Variables
      INTEGER     ISTART, IEND  , ISTEP , I     , J
```

```
      REAL        PREVAL, CURVAL
      LOGICAL     DONE  , BKWARD


      BKWARD = .FALSE.
      NITERS = IZERO

100   CONTINUE
         NITERS = NITERS + IONE

         IF ( BKWARD ) THEN
            ISTART = TOTELE
            IEND   = IONE
            ISTEP  = -1
            BKWARD = .FALSE.
         ELSE
            ISTART = IONE
            IEND   = TOTELE
            ISTEP  = IONE
C            BKWARD = .TRUE.
         END IF

         DO 300 I = ISTART, IEND, ISTEP
            PREVAL = X(I)
            CURVAL = B(I)
            DO 200 J = IONE, SYSINX(HEADER,I)
               CURVAL = CURVAL + SYSMAT(J,I) * X(SYSINX(J,I))
200         CONTINUE

            CURVAL = PREVAL + RELAXA * (CURVAL - PREVAL)

            RESVAL(I) = ABS ( CURVAL - PREVAL )
            X(I)      = CURVAL
300      CONTINUE

         CALL ERESID ( RMETHD, TOTELE, RESVAL, X     , BIGRES )

         CONVER = BIGRES .LE. TOLVAL
         DONE   = (NITERS .GE. MAXITR) .OR.
     @            (CONVER .AND. (.NOT. TOMITR))

         CALL SWAP ( X, 'E' )

      IF ( .NOT. DONE ) GOTO 100

      RETURN
      END
```

C.3 Diagonally Preconditioned Conjugate Gradient Solver

```
C-----------------------------------------------------------------------
C   Subroutine  ESOLVE
C
C   Author          C. Bailey
c                   K. McManus 17th November 1993
C                   Centre for Numerical Modelling & Process Analysis
C                   University of Greenwich, London, England.
C
C   Date            22 June 1992.
C
C   Description :   Solves the system Ax=B using the conjugate gradient
C                   method.
C   Variables :
C  IN   BANWID      -     Bandwidth
C  IN   TOTNOD      -     Total number of unknowns
C  IN   SYSINX      -     Index for the systems matrix.
C  IN   A           -     Systems matrix.
C  IN   B           -     Load vector.
C  IN   TOLVAL      -     Tolerance.
C  I/O  X           -     Unknown values.
C-----------------------------------------------------------------------
      SUBROUTINE ESOLVE ( BANWID, TOTNOD, SYSINX, A      , B      ,
     @                    TOLVAL, X      , MAXITR)

      INTEGER      BANWID, TOTNOD, MAXITR
      INTEGER      SYSINX(1:BANWID,1:TOTNOD)
      REAL         A      (1:BANWID,1:TOTNOD)
      REAL         B      (1:TOTNOD), X     (1:TOTNOD)
      REAL         TOLVAL

C     Local variables
      INTEGER      MAXBAN, MAXNOD, I      , J      , ITER
      PARAMETER    (MAXBAN = 10, MAXNOD = 500)
      REAL*8       PRCONA(1:MAXBAN,1:MAXNOD), PRCONB(1:MAXNOD)
      REAL*8       PRCONX(1:MAXNOD), OLDX  (1:MAXNOD)
      REAL*8       RESID (1:MAXNOD), U     (1:MAXNOD)
      REAL*8       P      (1:MAXNOD), B1    (1:MAXNOD)
      REAL*8       BIGDEV, ALPHAK, BETAK , DENOM , DENOM1
      REAL*8       RHOK, RHOKP
      LOGICAL      CONVER


      DO 2 I = 1, TOTNOD
         OLDX(I) = X(I)
         B1(I) = B(I) - A(1,I) * X(I)
         DO 3 J = 2, SYSINX(1,I)
            B1(I) = B1(I) - A(J,I) * X(SYSINX(J,I))
```

```
3        CONTINUE
2     CONTINUE


C==== Set up Pre-Conditioned matrix and vectors.
C==== Using diagonal scaling.

      DO 10 I = 1, TOTNOD
         X(I) = 0.0
         PRCONX(I) = 0.0
         PRCONB(I) = B1(I) / SQRT(A(1,I))
         DO 5 J = 2, SYSINX(1,I)
            PRCONA(J,I) = A(J,I) / SQRT( A(1,I) * A(1,SYSINX(J,I)))
5        CONTINUE
         PRCONA(1,I) = 1.0
10    CONTINUE


C==== Set up RESID, P .

      DENOM1 = 0.0
      DO 40 I = 1, TOTNOD
         RESID(I) = PRCONB(I)
         P(I) = RESID(I)
         DENOM1 = DENOM1 + (RESID(I) ** 2)
40    CONTINUE
      CALL GSUMD ( DENOM1 )
      IF ( DENOM1 .LE. 0.0 ) RETURN
      RHOK = DENOM1

C===== Start iteration cycle =====

      ITER = 0
      CONVER = .FALSE.
1001  ITER = ITER + 1
        CALL SWAP ( P, 'DN' )

C===== Calculate U(I) ======

         DO 60 I = 1, TOTNOD
            U(I) = P(I)
            DO 65 J = 2, SYSINX(1,I)
               U(I) = U(I) + PRCONA(J,I) * P(SYSINX(J,I))
65          CONTINUE
60       CONTINUE

C===== Calculate ALPHAK ======

         DENOM = 0.0
```

```
          DO 70 I = 1, TOTNOD
             DENOM = DENOM + P(I) * U(I)
70        CONTINUE
          CALL GSUMD ( DENOM )
          IF ( DENOM .LE. 0.0 ) THEN
             ALPHAK = 0.0
          ELSE
             ALPHAK = RHOK / DENOM
          END IF

C===== Calculate PRCONX and RESID at this iteration.

          RHOKP = 0.0
          DO 90 I = 1, TOTNOD
             PRCONX(I) = PRCONX(I) + ALPHAK * P(I)
             RESID(I) = RESID(I) - ALPHAK * U(I)
             RHOKP = RHOKP + (RESID(I) ** 2)
90        CONTINUE
          CALL GSUMD ( RHOKP )

C===== Calculate BETAK and P at this iteration =====

          IF ( RHOK .LE. 0.0 ) THEN
             BETAK = 0.0
          ELSE
             BETAK = RHOKP / RHOK
          END IF

          DO 130 I = 1, TOTNOD
             P(I) = RESID(I) + BETAK * P(I)
130       CONTINUE

C===== Calculate the residual norm.

          BIGDEV = SQRT ( RHOK / DENOM1 )

C===== Check to see if convergence has been achieved =====

          IF ((BIGDEV.GT.TOLVAL).AND.(ITER.LT.MAXITR))  THEN
            RHOK = RHOKP
            GOTO 1001
          END IF

C====== Calculate X from PRCONX.

       DO 500 I = 1, TOTNOD
          X(I) = PRCONX(I) / SQRT(A(1,I))
          X(I) = X(I) + OLDX(I)
          IF ( ABS(X(I)) .LT. 1.E-8 ) X(I) = 0
```

```
500   CONTINUE

      CALL SWAP ( X, 'N' )

      RETURN
      END
```

Appendix D

Modified Parallel Iterative Solvers

The codes listed here are the parallel Jacobi and conjugate gradient solvers that have been modified to provide improved parallel speed-up. The nature of the modifications is discussed in Section 5.4.

D.1 Modified Jacobi Solver

```
C-----------------------------------------------------------------------
C  Subroutine  JACOBI  (JACOBI) scheme
C
C
C  Author       : P.Chow  23rd March 1989
C                 K. McManus 23rd September 1993
C                 Centre for Numerical Modelling & Process Analysis
C                 University of Greenwich, London, England.
C
C  Description : Solve Ax = b using Jacobi iterative scheme.
C
C  Variables   :
C IN   RMETHD  -  Residual method.
C IN   TOLVAL  -  Tolerance value.
C IN   MAXITR  -  Maximum number of iteration.
C IN   TOMITR  -  To maximum iteration.
C IN   TOTELP  -  Total number of grid points per element.
C IN   TOTELE  -  Total number of element.
C IN   SYSINX  -  System matrix index.
C IN   SYSMAT  -  System matrix A.
C IN   B       -  B vector.
```

```
C I&O  X       -  Solving variable X.
C OUT  RESVAL  -  Residual values.
C OUT  NITERS  -  Number of iteration taken.
C OUT  BIGRES  -  Biggest residual value.
C OUT  CONVER  -  Convergent indicator.
C WSP  OX      -  Old X value (work space).
C
C-----------------------------------------------------------------------
      SUBROUTINE JACOBI ( RMETHD, TOLVAL, MAXITR, TOMITR, TOTELP,
     @                    TOTELE, SYSINX, SYSMAT, B     , X     ,
     @                    RESVAL, NITERS, BIGRES, CONVER, OX     )

      INTEGER     RMETHD, MAXITR, TOTELP, TOTELE, NITERS
      INTEGER     SYSINX(0:TOTELP,1:TOTELE)
      REAL        TOLVAL, BIGRES
      REAL        SYSMAT(1:TOTELP,1:TOTELE)
      REAL        B     (1:TOTELE), X     (1:TOTELE)
      REAL        RESVAL(1:TOTELE), OX    (1:TOTELE)
      LOGICAL     TOMITR, CONVER

C  Commons
      INCLUDE 'puifs.inc'

C  Local Constants
      INTEGER     HEADER, IZERO , IONE
      PARAMETER   ( HEADER = 0, IZERO  = 0, IONE   = 1 )

C  Local Variables
      INTEGER     ISTART, IEND  , ISTEP , I     , J
      LOGICAL     DONE

      NITERS = IZERO

      DO WHILE ( .NOT. DONE )
          NITERS = NITERS + IONE

C          DO 150 I = IONE, TOTELE - operate locally on the overlap
          DO 150 I = IONE, XTOTEL
             OX(I) = X(I)
150       CONTINUE

          DO 300 I = IONE, TOTELE
             X(I) = B(I)
             DO 200 J = IONE, SYSINX(HEADER,I)
                X(I) = X(I) + SYSMAT(J,I) * OX(SYSINX(J,I))
200          CONTINUE
300       CONTINUE

          IF ( TOMITR ) THEN
```

```
         DONE = (NITERS .GE. MAXITR)
      ELSE
        DO I = 1, TOTELE
          RESVAL(I) = ABS ( X(I) - OX(I) )
        END DO
        CALL ERESID ( RMETHD, TOTELE, RESVAL, X      , BIGRES )
        DONE = (NITERS .GE. MAXITR) .OR. (BIGRES .LE. TOLVAL)
      END IF

      CALL SWAP ( X, 'E' )

    END DO

    IF ( TOMITR ) THEN
      DO I = 1, TOTELE
        RESVAL(I) = ABS ( X(I) - OX(I) )
      END DO
      CALL ERESID ( RMETHD, TOTELE, RESVAL, X      , BIGRES )
    END IF

    RETURN
    END
```

D.2 Modified Diagonally Preconditioned Conjugate Gradient Solver

```
C-----------------------------------------------------------------------
C   Subroutine  ESOLVE
C
C   Author          C. Bailey
c                   K. McManus 10th July 1995
C                   Centre for Numerical Modelling & Process Analysis
C                   University of Greenwich, London, England.
C
C   Date            22 June 1992.
C
C   Amendments.
C
C   Description : Solves the system Ax=B using the conjugate gradient
C                 method.
C   Variables :
C  IN   BANWID     -    Bandwidth
C  IN   TOTNOD     -    Total number of unknowns
C  IN   SYSINX     -    Index for the systems matrix.
C  IN   A          -    Systems matrix.
C  IN   B          -    Load vector.
C  IN   TOLVAL     -    Tolerance.
C  I/O  X          -    Unknown values.
C-----------------------------------------------------------------------
      SUBROUTINE ESOLVE ( BANWID, TOTNOD, SYSINX, A      , B      ,
     @                    TOLVAL, X      , MAXITR)

      INTEGER      BANWID, TOTNOD, MAXITR
      INTEGER      SYSINX(1:BANWID,1:TOTNOD)
      REAL         A      (1:BANWID,1:TOTNOD)
      REAL         B      (1:TOTNOD), X     (1:TOTNOD)
      REAL         TOLVAL

C   Local variables
      INTEGER      MAXBAN, MAXNOD, I      , J      , ITER
      PARAMETER    (MAXBAN = 10, MAXNOD = 500)
      REAL*8       PRCONA(1:MAXBAN,1:MAXNOD), PRCONB(1:MAXNOD)
      REAL*8       PRCONX(1:MAXNOD), OLDX  (1:MAXNOD)
      REAL*8       RESID (1:MAXNOD), U     (1:MAXNOD)
      REAL*8       P      (1:MAXNOD), B1    (1:MAXNOD)
      REAL*8       BIGDEV, ALPHAK, BETAK , DENOM , DENOM1
      REAL*8       RHOK, RHOKP
      REAL*8       UU, RESIDU
      LOGICAL      CONVER
```

```
      DO 2 I = 1, TOTNOD
         OLDX(I) = X(I)
         B1(I) = B(I) - A(1,I) * X(I)
         DO 3 J = 2, SYSINX(1,I)
            B1(I) = B1(I) - A(J,I) * X(SYSINX(J,I))
3        CONTINUE
2     CONTINUE


C==== Set up Pre-Conditioned matrix and vectors.
C==== Using diagonal scaling.

      DO 10 I = 1, TOTNOD
         X(I) = 0.0
         PRCONX(I) = 0.0
         PRCONB(I) = B1(I) / SQRT(A(1,I))
         DO 5 J = 2, SYSINX(1,I)
            PRCONA(J,I) = A(J,I) / SQRT( A(1,I) * A(1,SYSINX(J,I)))
5        CONTINUE
         PRCONA(1,I) = 1.0
10    CONTINUE


C==== Set up RESID, P .

      DENOM1 = 0.0
      DO 40 I = 1, TOTNOD
         RESID(I) = PRCONB(I)
         P(I) = RESID(I)
         DENOM1 = DENOM1 + (RESID(I) ** 2)
40    CONTINUE

      CALL GSUMD ( DENOM1 )

      IF ( DENOM1 .LE. 0.0 ) RETURN
      RHOK = DENOM1

C===== Start iteration cycle =====

      ITER = 0
      CONVER = .FALSE.
1001  ITER = ITER + 1
      CALL SWAP ( P, 'DN' )

C===== Calculate U(I) ======

         DO 60 I = 1, TOTNOD
            U(I) = P(I)
            DO 65 J = 2, SYSINX(1,I)
```

```
                U(I) = U(I) + PRCONA(J,I) * P(SYSINX(J,I))
65           CONTINUE
60        CONTINUE

C===== Calculate ALPHAK ======

          DENOM = 0.0
          UU = 0.0
          RESIDU = 0.0
          DO 70 I = 1, TOTNOD
             DENOM = DENOM + P(I) * U(I)
             UU = UU + U(I)*U(I)
             RESIDU = RESIDU + RESID(I)*U(I)
70        CONTINUE
          CALL GSUMD3 ( DENOM, UU, RESIDU )
          IF ( DENOM .LE. 0.0 ) THEN
             ALPHAK = 0.0
          ELSE
             ALPHAK = RHOK / DENOM
          END IF
          RHOKP = RHOK + ALPHAK * ( ALPHAK*UU - 2*RESIDU )

C===== Calculate PRCONX and RESID at this iteration.

          RHOKP = 0.0
          DO 90 I = 1, TOTNOD
             PRCONX(I) = PRCONX(I) + ALPHAK * P(I)
             RESID(I) = RESID(I) - ALPHAK * U(I)
90        CONTINUE

C===== Calculate BETAK and P at this iteration =====

          IF ( RHOK .LE. 0.0 ) THEN
             BETAK = 0.0
          ELSE
             BETAK = RHOKP / RHOK
          END IF

          DO 130 I = 1, TOTNOD
             P(I) = RESID(I) + BETAK * P(I)
130       CONTINUE

C===== Calculate the residual norm.

          BIGDEV = SQRT ( RHOK / DENOM1 )

C===== Check to see if convergence has been achieved =====

          IF ((BIGDEV.GT.TOLVAL).AND.(ITER.LT.MAXITR))  THEN
```

```
            RHOK = RHOKP
            GOTO 1001
          END IF

C====== Calculate X from PRCONX.

      DO 500 I = 1, TOTNOD
         X(I) = PRCONX(I) / SQRT(A(1,I))
         X(I) = X(I) + OLDX(I)
         IF ( ABS(X(I)) .LT. 1.E-8 ) X(I) = 0
500   CONTINUE

      CALL SWAP ( X, 'N' )

      RETURN
      END
```

Appendix E

Asynchronous Parallel Iterative Solvers

To achieve an improved efficiency through the use of asynchronous communications to overlap communication and calculation, the loop structure has to be split to operate firstly on the variables required for communication and then on the rest of the sub-domain. To succeed this requires that the sub-domain core has been renumbered with the dependent elements (grid points), i.e. those that are required for communication to neighbouring sub-domains, being numbered before the rest of the sub-domain. This is discussed in Section 5.4.5 A new variable `NDEPEL` contained in puifs.inc records the number of dependent elements. The element loop can now loop over `NDEPEL` elements, initiate communication, loop over `NDEPEL+1` to `TOTELE` and then synchronise the communication. Listed here are asynchronous versions of the Jacobi and conjugate gradient parallel solvers. The techniques illustrated here can be applied to many other code structures.

E.1 Asynchronous Jacobi Solver

```
C-----------------------------------------------------------------------
C  Subroutine  JACOBI  (JACOBI) scheme
```

```
C
C
C  Author       : P.Chow  23rd March 1989
C                  K. McManus 21st July 1995
C                  Centre for Numerical Modelling & Process Analysis
C                  University of Greenwich, London, England.
C
C  Description  : Solve Ax = b using JOR iterative scheme.
C
C  Variables    :
C IN   RMETHD  -  Residual method.
C IN   RELAXA  -  Relaxation value.
C IN   TOLVAL  -  Tolerance value.
C IN   MAXITR  -  Maximum number of iteration.
C IN   TOMITR  -  To maximum iteration.
C IN   TOTELP  -  Total number of grid points per element.
C IN   TOTELE  -  Total number of element.
C IN   SYSINX  -  System matrix index.
C IN   SYSMAT  -  System matrix A.
C IN   B       -  B vector.
C I&O  X       -  Solving variable X.
C OUT  RESVAL  -  Residual values.
C OUT  NITERS  -  Number of iteration taken.
C OUT  BIGRES  -  Biggest residual value.
C OUT  CONVER  -  Convergent indicater.
C WSP  OX      -  Old X value (work space).
C
C-----------------------------------------------------------------------
      SUBROUTINE JACOBI ( RMETHD, RELAXA, TOLVAL, MAXITR, TOMITR,
     @                    TOTELP, TOTELE, SYSINX, SYSMAT, B     ,
     @                    X     , RESVAL, NITERS, BIGRES, CONVER,
     @                    OX     )

      INTEGER     RMETHD, MAXITR, TOTELP, TOTELE, NITERS
      INTEGER     SYSINX(0:TOTELP,1:TOTELE)
      REAL        RELAXA, TOLVAL, BIGRES
      REAL        SYSMAT(1:TOTELP,1:TOTELE)
      REAL        B     (1:TOTELE), X     (1:TOTELE)
      REAL        RESVAL(1:TOTELE), OX    (1:TOTELE)
      LOGICAL     TOMITR, CONVER

C  Commons
      INCLUDE 'puifs.inc'

C  Local Constants
      INTEGER     HEADER, IZERO , IONE
      PARAMETER   ( HEADER = 0, IZERO  = 0, IONE   = 1 )

C  Local Variables
```

```
INTEGER     ISTART, IEND  , ISTEP , I     , J     , ID
LOGICAL     DONE


NITERS = IZERO
DONE = .FALSE.

DO WHILE ( .NOT. DONE )

    NITERS = NITERS + IONE

   DO I = IONE, XTOTEL
      OX(I) = X(I)
   END DO

   DO I = IONE, NDEPEL
      X(I) = B(I)
      DO J = IONE, SYSINX(HEADER,I)
         X(I) = X(I) + SYSMAT(J,I) * OX(SYSINX(J,I))
      END DO
   END DO

   CALL ASWAP ( X, 'E', ID )

   DO I = NDEPEL+IONE, TOTELE
      X(I) = B(I)
      DO J = IONE, SYSINX(HEADER,I)
         X(I) = X(I) + SYSMAT(J,I) * OX(SYSINX(J,I))
      END DO
   END DO

   CALL SYNC ( ID )

   IF ( TOMITR ) THEN
      DONE = (NITERS .GE. MAXITR)
   ELSE
      DO I = 1, TOTELE
         RESVAL(I) = ABS ( X(I) - OX(I) )
      END DO
      CALL ERESID ( RMETHD, TOTELE, RESVAL, X     , BIGRES )
      DONE = (NITERS .GE. MAXITR) .OR. (BIGRES .LE. TOLVAL)
   END IF

END DO

IF ( TOMITR ) THEN
   DO I = 1, TOTELE
      RESVAL(I) = ABS ( X(I) - OX(I) )
   END DO
```

```
         CALL ERESID ( RMETHD, TOTELE, RESVAL, X      , BIGRES )
      END IF

      RETURN
      END
```

E.2 Asynchronous Diagonally Preconditioned Conjugate Gradient Solver

```
C-----------------------------------------------------------------------
C   Subroutine  ESOLVE
C
C   Author          C. Bailey
c                   K. McManus 14th August 1995
C                   Centre for Numerical Modelling & Process Analysis
C                   University of Greenwich, London, England.
C
C   Date            22 June 1992.
C
C   Amendments.
C
C   Description : Solves the system Ax=B using the conjugate gradient
C                 method.
C   Variables :
C  IN   BANWID     -    Bandwidth
C  IN   TOTNOD     -    Total number of unknowns
C  IN   SYSINX     -    Index for the systems matrix.
C  IN   A          -    Systems matrix.
C  IN   B          -    Load vector.
C  IN   TOLVAL     -    Tolerance.
C  I/O  X          -    Unknown values.
C-----------------------------------------------------------------------
      SUBROUTINE ESOLVE ( BANWID, TOTNOD, SYSINX, A      , B      ,
     @                    TOLVAL, X      , MAXITR)

      INTEGER      BANWID, TOTNOD, MAXITR
      INTEGER      SYSINX(1:BANWID,1:TOTNOD)
      REAL         A      (1:BANWID,1:TOTNOD)
      REAL         B      (1:TOTNOD), X     (1:TOTNOD)
      REAL         TOLVAL

C   Local variables
      INTEGER      MAXBAN, MAXNOD, I      , J      , ITER, ID
      PARAMETER    (MAXBAN = 10, MAXNOD = 500)
      REAL*8       PRCONA(1:MAXBAN,1:MAXNOD), PRCONB(1:MAXNOD)
      REAL*8       PRCONX(1:MAXNOD), OLDX  (1:MAXNOD)
```

```
      REAL*8      RESID (1:MAXNOD), U     (1:MAXNOD)
      REAL*8      P     (1:MAXNOD), B1    (1:MAXNOD)
      REAL*8      BIGDEV, ALPHAK, BETAK , DENOM , DENOM1
      REAL*8      RHOK, RHOKP
      REAL*8      UU, RESIDU
      LOGICAL     CONVER


      DO 2 I = 1, TOTNOD
         OLDX(I) = X(I)
         B1(I) = B(I) - A(1,I) * X(I)
         DO 3 J = 2, SYSINX(1,I)
            B1(I) = B1(I) - A(J,I) * X(SYSINX(J,I))
3        CONTINUE
2     CONTINUE


C==== Set up Pre-Conditioned matrix and vectors.
C==== Using diagonal scaling.

      DO 10 I = 1, TOTNOD
         X(I) = 0.0
         PRCONX(I) = 0.0
         PRCONB(I) = B1(I) / SQRT(A(1,I))
         DO 5 J = 2, SYSINX(1,I)
            PRCONA(J,I) = A(J,I) / SQRT( A(1,I) * A(1,SYSINX(J,I)))
5        CONTINUE
         PRCONA(1,I) = 1.0
10    CONTINUE


C==== Set up RESID, P .

      DENOM1 = 0.0
      DO 40 I = 1, TOTNOD
         RESID(I) = PRCONB(I)
         P(I) = RESID(I)
         DENOM1 = DENOM1 + (RESID(I) ** 2)
40    CONTINUE

      CALL GSUMD ( DENOM1 )

      IF ( DENOM1 .LE. 0.0 ) RETURN
      RHOK = DENOM1

      CALL ASWAP ( P, 'DN', ID )

C===== Start iteration cycle =====
```

```
      ITER = 0
      CONVER = .FALSE.
1001  ITER = ITER + 1

C===== Calculate U(I) ======

         DO 60 I = NDEPGP+1, TOTNOD
            U(I) = P(I)
            DO 61 J = 2, SYSINX(1,I)
               U(I) = U(I) + PRCONA(J,I) * P(SYSINX(J,I))
61          CONTINUE
60       CONTINUE

         CALL SYNC ( ID )

         DO 62 I = 1, NDEPGP
            U(I) = P(I)
            DO 63 J = 2, SYSINX(1,I)
               U(I) = U(I) + PRCONA(J,I) * P(SYSINX(J,I))
63          CONTINUE
62       CONTINUE

C===== Calculate ALPHAK ======

         DENOM = 0.0
         UU = 0.0
         RESIDU = 0.0
         DO 70 I = 1, TOTNOD
            DENOM = DENOM + P(I) * U(I)
            UU = UU + U(I)*U(I)
            RESIDU = RESIDU + RESID(I)*U(I)
70       CONTINUE
         CALL GSUMD3 ( DENOM, UU, RESIDU )
         IF ( DENOM .LE. 0.0 ) THEN
            ALPHAK = 0.0
         ELSE
            ALPHAK = RHOK / DENOM
         END IF
         RHOKP = RHOK + ALPHAK * ( ALPHAK*UU - 2*RESIDU )

C===== Calculate PRCONX and RESID at this iteration.

         RHOKP = 0.0
         DO 90 I = 1, TOTNOD
            PRCONX(I) = PRCONX(I) + ALPHAK * P(I)
            RESID(I) = RESID(I) - ALPHAK * U(I)
90       CONTINUE

C===== Calculate BETAK at this iteration =====
```

```
         IF ( RHOK .LE. 0.0 ) THEN
            BETAK = 0.0
         ELSE
            BETAK = RHOKP / RHOK
         END IF

C===== Calculate the residual norm.

         BIGDEV = SQRT ( RHOK / DENOM1 )

C===== Check to see if convergence has been achieved =====

         IF ((BIGDEV.GT.TOLVAL).AND.(ITER.LT.MAXITR))  THEN
           RHOK = RHOKP

           DO 130 I = 1, NDEPGP
             P(I) = RESID(I) + BETAK * P(I)
130        CONTINUE

           CALL ASWAP ( P, 'DN', ID )

           DO 131 I = NDEPGP+1, TOTNOD
              P(I) = RESID(I) + BETAK * P(I)
131        CONTINUE

           GOTO 1001
         END IF

C====== Calculate X from PRCONX.

      DO 500 I = 1, TOTNOD
         X(I) = PRCONX(I) / SQRT(A(1,I))
         X(I) = X(I) + OLDX(I)
         IF ( ABS(X(I)) .LT. 1.E-8 ) X(I) = 0
500   CONTINUE

      CALL SWAP ( X, 'N' )

      RETURN
      END
```

Bibliography

[AG94] George S. Almasi and Allan Gottlieb. *Highly Parallel Computing, 2nd Edition.* Benjamin/Cummings, Redwood City, 1994.

[Amd67] G. M. Amdahl. Validity of the single-processor approach to achieving large scale computing capabilities. In *Proc AFIPS*, pages 483–485, 1967.

[BA92] Tevfik Bultan and Cevdet Aykanat. A new mapping heuristic based on mean field annealing. *Parallel and Distributed Computing*, 16:292–305, September 1992.

[Ban79] U. Bannerjee. *Speedup of Ordinary Programs.* PhD thesis, University of Illinois at Urbana Champaign, 1979.

[Ban88] U. Bannerjee. *Dependence Analysis for Supercomputing.* Kluwer Academic Publishers, 1988.

[BBC+94] Rchard Barrett, Michael Berry, Tony Chan, James Demmel, June Donato, Jack Dongarra, Victor Eijkhout, Roldan Pozo, Charles Romine, and Henk van der Vorst. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods.* Netlib, 1994.

[BBLS93] David Bailey, John Barton, Thomas Lasinski, and Horst Simon. The NAS parallel benchmarks. Technical Memorandum 103863, NASA, July 1993.

[BCG93] P. E. Bjørstad, W. M. Coughran, and E. Grosse. Parallel domain decomposition applied to coupled transport equations. In *Domain Decomposition Methods 7*, October 1993.

[Bom93] Erik Boman. Experiences on the KSR1 computer. Technical Report RNR-93-008, NAS Applied Research Branch, April 1993.

[BS93] S. T. Barnard and H. D. Simon. A fast multilevel implementation of recursive spectral bisection for partitioning unstructured problems. In *Proc 6th SIAM Conf, Parallel Processing for Scientific Computing*, pages 711–718, 1993.

[BT94] R. Battiti and G. Tecchiolli. The reactive tabu search. *ORSA Journal on Computing*, 6(2):126–140, 1994.

[CBB+94] B. M. Chapman, S. Benkner, R. Blasko, P. Brezany M. Egg, T. Fahringer, H. M. Gerndt, J. Hulman, B. Knaus, P. Kutschera, H. Moritsch, A. Schwald, V. Sipkova, and H. P. Zima. *VIENNA FORTRAN Compilation System Version 1.0 User's Guide.* University of Vienna, January 1994.

[CBCP92] M. Cross, C. Bailey, P. Chow, and K. Pericleous. Towards an integrated control volume unstructured mesh code for the simulation of all the macroscopic processes involved in shape casting. In *Numerical Methods in Industrial Forming Processes, (NUMIFORM 92)*, pages 787–792. Balkema, 1992.

[CDE+94] C. Clémençon, K. M. Decker, A. Endo, J. Fritscher, G. Jost, N. Masuda, A. Müller, R. Rühl, W. Sawyer, E. de Sturler, and B. J. N. Wylie. Application driven development of an integrated tool environment for distributed parallel processors. Technical Report CSCS-TR-94-01, CSCS-ETH, April 1994.

[CDJ95] Henri Casanova, Jack Dongarra, and Weicheng Jiang. The performance of PVM on MPP systems. Technical report, Dept of Computer Science, University of Tenessee, July 1995.

[CHA94] CHAM, Wimbledon, UK. *The Phoenics Reference Manual*, 1994.

[Che91] J. Chen. *The Numerical Solution of Complex Fluid Flow Phenomena.* PhD thesis, University of Leeds, 1991.

[Cho93] Peter Chow. *A Control Volume Unstructured Mesh Procedure for Convection-Diffusion Solidification Processes.* PhD thesis, University of Greenwich, 1993.

[CIJL94] M. Cross, C. S. Ierotheou, S. P. Johnson, and P. F. Leggett. CAPTools - semiautomatic parallelisation of mesh based computational mechanics codes. In *High Performance Computing and Networking*, volume II, pages 241–246. Springer Verlag, 1994.

[CJL+89] F. Cheng, J. W .Jaromczyk, J-R. Lin, S-S. Chang, and J-Y. Lu. A parallel mesh generation algorithm based on the vertex label assignment scheme. *International Journal for Numerical Methods in Engineering*, 28, 1989.

[CTHW91] Lyndon Clark, Arthur Trew, Neil Heywood, and Matthew White. CHIMP concepts. Technical Report EPCC-KTP-CHIMP-CONC 1.2, Edinburgh Parallel Computing Centre, June 1991.

[dC95] R. Dias da Cunha. Parallel preconditioned conjugate gradient methods on transputer networks. *Transputer Communications*, 1995. (submitted for publication).

[DD95] Jack J. Dongarra and Tom Dunigan. Message passing performance of various computers. Technical report, University of Tenessee and Oak Ridge National Laboratory, University of Tenessee and Oak Ridge National Laboratory, 1995.

[DeC89] Angel L. DeCegama. *The Technology of Parallel Processing*, volume 1. Prentice-Hall, 1989.

[DER93] Eduardo D'Azvedo, Victor Eijkhout, and Charles Romine. Reducing communication costs in the conjugate gradient algorithm on distributed memory multiprocessors. Technical Report CS-93-185, Lapack Working Note 56, Oak Ridge National Laboratory and The University of Tennessee, Knoxville, January 1993.

[DLD93] David Callahan David Levine and Jack Dongarra. A comparative study of automatic vectorising compilers. *Parallel Computing*, 17:1223–1244, 1993.

[DMM95] Ralf Diekman, Derk Meyer, and Burkhard Monien. Parallel decomposition of unstructured FEM-meshes. In *Parallel Algorithms for Irregularly Structured Problems*. Springer, September 1995.

[Far88] C. Farhat. A simple and efficient automatic FEM domain decomposer. *Computers and Structures*, 28:579–602, 1988.

[Far89] C. Farhat. On the mapping of massively parallel processors onto finite element graphs. *Computers and Structures*, 32(2):347–353, 1989.

[FBCL91] Y. D. Fryer, C. Bailey, M. Cross, and C-H. Lai. A control volume procedure for solving the elastic stress-strain equations on an unstructured mesh. *Appl. Math. Modelling*, 15, November 1991.

[FFL93] Charbel Farhat, Loula Fezoui, and Stéphane Lanteri. Two-dimensional viscous flow computations on the connection machine: Unstructured meshes, upwind schemes and massively parallel computations. *Computer Methods in Applied Mechanics and Engineering*, 102:61–68, 1993.

[FG94] R. F. Fowler and C. Greenough. Ralpar- ral mesh partitioning program version 1.1. Internal report, Rutherford Appleton Laboratory, May 1994.

[FJL+88] G. C. Fox, M. Johnson, G. Lyzenga, S. Otto, J. Salmon, and D. Walker. *Solving Problems on Concurrent Processors*, volume I. Prentice Hall, Englewood Cliffs, NJ, 1988.

[Fly72] Michael J. Flynn. Some computer organizations and their effectiveness. *IEEE Transactions on Computers*, C-21:948–960, 1972.

[For94] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard.* University of Tenessee, May 1994.

[Fox88] G. C. Fox. *Numerical Algorithms for Modern Parallel Computers.* Springer-Verlag, 1988.

[FR94] N. Floros and J. S. Reeve. Domain decomposition tool. Technical report, Southampton HPC Centre, 1994.

[Fry93] Y. D. Fryer. *A Control Volume Unstructured Grid Approach to the Solution of the Elastic Stress-Strain Equations.* PhD thesis, University of Greenwich, 1993.

[FWM94] G. C. Fox, R. D. Wiliams, and P. C. Messina. *Parallel Computing Works.* Morgan Kaufmann, 1994.

[FXR92] C. Farhat and F. Xavier-Roux. An unconventional domain decomposition method for an efficient parallel solution of large-scale finite element systems. *SIAM J. Sci. Stat. Comp.*, 13(1):379–396, January 1992.

[GBD+94] Al Geist, A. Beguelin, J. Dongarra, Jiang Weicheng, Robert Manchek, and Vaidy Sunderam. *PVM: Parallel Virtual Machine - A Users' Guide and Tutorial for Networked Parallel Computing.* MIT Press, 1994.

[GCC+93] E. R Galea, A. Chan, M. Cross, N. Hoffman, C. Ierotheou, S. Johnson, and K. Pericleous. Application of a parallel CFD code to large scale practical systems. In *Parallel Computational Fluid Dynamics '92*, pages 147–155. Elsevier Science Publishers B.V., 1993.

[GF94] C. Greenough and R. F. Fowler. Partitioning methods for unstructured finite element meshes. Internal report, Rutherford Appleton Laboratory, March 1994.

[GHPW90] G. A. Geist, M. T. Heath, B. W. Peyton, and P.H. Worley. A users' guide to PICL a portable instrumented communication library. Technical Report ORNL/TM-11616, Oak Ridge National Laboratory, October 1990.

[GL89] Gene H. Golub and Charles F. Van Loan. *Matrix Computations: Second Edition.* John Hopkins University Press, Baltimore and London, 1989.

[Glo89] F. Glover. Tabu search - part i. *ORSA Journal on Computing*, 1(3):190–260, 1989.

[Glo90] F. Glover. Tabu search - part ii. *ORSA Journal on Computing*, 2(1):4–32, 1990.

[GMD95a] GMD. Camas-link. *ESPRIT EUROPORT - 1 Newsletter*, 5:1–2, August 1995.

[GMD95b] GMD. Parallel phoenics. *ESPRIT EUROPORT - 1 Newsletter*, 3:1–2, July 1995.

[GWZ95] P. W. Grant, M. F. Webster, and X. Zhang. Solving computational fluid dynamics problems on unstructured grids with distributed parallel processing. In *Parallel Algorithms for Irregularly Structured Problems.* Springer, September 1995.

[Har94] Haritaoğlu, İ and Aykanat, C. An efficient mapping heuristic fo mesh-connected parallel architectures based on mean field annealing. In *Parallel Processing: CONPAR 94 - VAPP VI*, pages 820–831. Springer Verlag, 1994.

[HB84] Kai Hwang and Fayé A Briggs. *Computer Architecture and Parallel Processing.* McGraw-Hill, 1984.

[Hei94] R. Heimes. pV3: A distributed system for large-scale unstedy CFD visualization. Technical report, AIAA, 1994.

[Hem91] R. Hempel. The ANL/GMD macros (PARMACS) in FORTRAN for portable parallel programming using the message passing programming model. User's guide and reference manual, Gesellschaft für Mathematik und Datenverarbeitung mbH, November 1991.

[Hil94] Jonathon M. D. Hill. An introduction to the data-parallel paradigm. Sel-hpc course material, LPAC, 1994.

[HJ81] R. W. Hockney and C. R. Jessope. *Parallel Computers: architectures, programming and algorithms.* Adam Hilger, Bristol, 1981.

[HJ88] R. W. Hockney and C. R. Jessope. *Parallel Computers 2: architectures, programming and algorithms.* Adam Hilger, Bristol, 1988.

[HJ94] D. C. Hodgeson and P. K. Jimak. Parallel generation of partitioned, unstructured meshes. Report 94.19, University of Leeds, School of Computer Studies Research, June 1994.

[HL92] B. Hendrickson and R. Leland. An improved spectral graph partitioning algorithm for mapping parallel computations. Tech. rep. sand 92-1460, Sandia National Labs, Albuquerque, NM, 1992.

[HL93] B. Hendrickson and R. Leland. A multilevel algorithm for partitioning graphs. Tech. rep. sand 93-1301, Sandia National Labs, Albuquerque, NM, 1993.

[Hoa86] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice-Hall, 1986.

[HS92] Steven W Hammond and Robert Schreiber. Mapping unstructured grid problems to the connection machine. In Piyush Mehrota, Joel Saltz, and Robert Voigt, editors, *Unstructured Scientific Computation on Scalable Multiprocessors*, pages 11–29. MIT Press, 1992.

[Ier90] Constantinos S Ierotheou. *The Simulation of Fluid Flow Processes Using Vector Processors.* PhD thesis, Thames Polytechnic, 1990.

[IFB95] C. S. Ierotheou, C. R. Forsey, and U. Block. Parallelisation of a novel 3d hybrid structured-unstructured grid CFD production code. In *High-Performance Computing and Networking*, pages 831–836. Springer, May 1995.

[Inm89a] Inmos. *Transputer Applications Notebook: Architecture and Software*, 1989.

[Inm89b] Inmos. *Transputer Applications Notebook: Systems and Performance*, 1989.

[Inm89c] Inmos. *The Transputer Databook*, 1989.

[Inm92] Inmos. *ANSI C Toolset User Guide*, 1992.

[JAC92] S. P. Johnson, F. Ali, and M. Cross. Parallelising the FAMCALC FEA code. Report, Centre for Numerical Modelling and Process Analysis, February 1992.

[JC91] S. P. Johnson and M. Cross. Mapping structured grid three-dimensional CFD codes onto parallel architectures. *Appl. Math. Modelling*, 15:948–960, August 1991.

[JCI+94] S. P. Johnson, M. Cross, C. S. Ierotheou, P. F. Leggett, and A. T. J. Marsh. Computer aided parallelisation tools (CAPTools) for real CFD applications. In *Proc. Parallel CFD'94*. North Holland, 1994. in press.

[JICL94] S. P. Johnson, C. S. Ierotheou, M. Cross, and P. F. Leggett. User interaction and symbolic extensions to dependence analysis. In *Parallel Processing: CONPAR 94 - VAPP VI*, pages 725–736. Springer Verlag, 1994.

[Joh92] S. P. Johnson. *Mapping Numerical Software onto Distributed Memory Parallel Systems*. PhD thesis, University of Greenwich, 1992.

[Jon94] B. W. Jones. *Mapping Unstructured Mesh Codes onto Local Memory Parallel Architectures*. PhD thesis, School of Maths., University of Greenwich, London SE18 6PF, UK, 1994.

[KJV83] S. Kirkpatrick, C. D. Gelatt Jr., and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220:671–680, 1983.

[KK95] George Karypis and Vipin Kumar. Multilevel k-way partitioning scheme for irregular graphs. Technical Report 95-064, Department of Computer Science, University of Minnesota, August 1995.

[KL70] B. W. Kernighan and S. Lin. An efficient heuristic procedure for partitioning graphs. *The Bell System Technical Journal*, pages 291–307, Febuary 1970.

[Kri89] E. V. Krishnamurthy. *Parallel Processing Principles and Practice.* Addison Wesley, 1989.

[Kro63] G. Kron. *Diakoptics: The Piecewise Solution of Large-Scale Systems.* MacDonald & Co., London, 1963.

[KX93] David E. Keyes and Jinchao Xu, editors. *Domain Decomposition Methods in Scientific and Engineering Computing.* AMA, 1993. Proceedings of the 7th International Conference on Domain Decomposition.

[Lai95] C-H. Lai. On domain decomposition and mapping issues for massively parallel computing. Report P95/IM/04, University of Greenwich, 1995.

[Law94] Peter James Lawrence. *Mesh Generation by Domain Bisection.* PhD thesis, University of Greenwich, 1994.

[LC90] P. F. Leggett and M. Cross. Parallel processing approaches to view factor calculation. Report, Thames Polytechnic, 1990.

[LL88] C-H. Lai and H. M. Liddell. Preconditioned conjugate gradient methods on the DAP. In *The Mathematics of Finite Elements and Applications VI*, pages 145–156. Academic Press, 1988.

[LP92] Wei Li and Keshav Pingali. Access normalisation: Loop restructuring for NUMA compilers. Technical Report TR92-1278, Cornell University, 1992.

[MF95] Grant McFarland and Michael Flynn. Limits of scaling MOSFETs. Technical Report CSL-TR-95-662, Stanford University, January 1995.

[MJ95] D R McCarthy and W R Jones. Adaptive domain decomposition and parallel CFD. In *Parallel Computational Fluid Dynamics: New Trends and Advances*, pages 31–40. Elsevier Science Publishers B.V., 1995.

[MR93] Richard Miller and Joy Reed. The oxford BSP library users' guide version 1.0. Technical report, Oxford Computing Laboratory, 1993.

[MSS+88] R. Mirchandaney, J. Saltz, R. Smith, D. Nicol, and K. Crowley. Principles of runtime support for parallel processors. In *Proc. Second Int. Conf. on Supercomputing*, July 1988.

[MSSP88] W. J. Mincowycz, E. M. Sparrow, G. E. Schneider, and R. H. Pletcher. *Handbook of Numerical Heat Transfer.* Wiley, 1988.

[MWC+95] K. McManus, C. Walshaw, M. Cross, P. Leggett, and S. Johnson. Evaluation of the JOSTLE mesh partitioning code for practical multiphysics applications. In *Proceedings PCFD'95*, 1995. submitted for publication.

[NM93] L. M. Ni and P. K. McKinley. A survey of wormhole routing techniques in direct networks. *IEEE Computer*, pages 62–76, Feb 1993.

[Par82] Dennis Parkinson. Practical parallel processors and their uses. In D. J. Evans, editor, *Parallel Processing Systems*, pages 216–236. Cambridge University Press, 1982.

[Par92] ParaSoft Corporation, Pasadena, CA, USA. *Express: Building Parallel and Distributed Programs*, 1992.

[Pat80] S V Patankar. *Numerical Heat Transfer and Fluid Flow.* Hemisphere, Washington DC, 1980.

[PS92] A. J. Peace and A. J. Shaw. The modelling of aerodynamic flows by the solution of the eulaer equations on mixed polyhedra grids. *Int. J. Numerical Methods in Engineering*, 35:2003–2029, 1992.

[PSL89] A. Pothen, H. D. Simon, and K. P. Liu. Partitioning sparse matrices with eigenvectors of graphs. Technical Report RNR-89-009, NASA Ames Research Centre, July 1989.

[RC82] C. M. Rhie and W. L. Chow. A numerical study of the turbulent flow past an isolated airfoil with trailing edge separation. *JAIAA*, 21:1525–1532, 1982.

[Ric95] H. Richardson. High performance fortran: history, overview and current developments. Technical report, Thinking Machines Corporation, March 1995.

[RL90] Guy Robinson and Richard Lonsdale. Fluid dynamics in parallel using an unstructured mesh. Internal report, UKAEA, April 1990.

[Rod82] Garry Rodrigue. *Parallel Computations.* Academic Press, New York, 1982.

[RVD93] D. Roose and R. Van Driessche. Distributed memory parallel computers and computational fluid dynamics. TW Report 186, Department of Computer Science, Katholieke Universiteit Leuven, Belgium, March 1993.

[SE87] Ponnuswamy Saddayapan and Fikret Ercal. Cluster partitioning approaches to mapping parallel programs onto a hypercube. *IEEE Transactions on Computers*, C-36(12):1408–1421, 1987.

[SER90] P. Saddayapan, F. Ercal, and J. Ramanujam. Cluster partitioning approaches to mapping parallel programs onto a hypercube. *Parallel Computing*, 13:1–16, 1990.

[She94] J. R. Shewchuck. An introduction to the conjugate gradient method without the agonizing pain. Technical Report CMU-CS-94-125, Carnegie Mellon University, March 1994.

[Sim91] H. D. Simon. Partitioning of unstructured problems for parallel processing. *Computing Systems in Engineering*, 2(2/3):135–148, 1991.

[Smi90] Burton Smith. The end of architecture. Keynote Address, 17th Annual Symposium on Computer Architecture, Washington, May 1990.

[SR87] G. E. Schneider and M. J. Raw. Control volume finite-element method for heat transfer and fluid flow using colocated variables - 1. computational procedure. *Numerical Heat Transfer*, 2:363–390, 1987.

[Sun94] Sun Microsystems, Mountain View, CA. *Fortran 3.0.1 User's Guide*, August 1994.

[TW91] Arthur Trew and Greg Wilson, editors. *Past, Present, Parallel: A Survey of Available Parallel Computer Systems.* Springer-Verlag, 1991.

[Val90] Leslie G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, pages 103–111, August 1990.

[VCM87] V. R. Voller, M. Cross, and N. C. Markatos. An enthalpy method for convection/diffusion phase change. *International Journal for Numerical Mehtods in Engineering*, 24:271–284, 1987.

[vdS94] Aad J. van der Steen. An overview of recent supercomuters. Technical report, Stichting Nationale Computer Faciliteiten, September 1994. 4th edition.

[vH92] R v Hanxleden. Compiler support for machine independent parallelisation of irregular problems. Technical Report CRPC-TR92301-S, Center for Research on Parallel Computation, Rice University, November 1992.

[VK95] D. Vanderstraeten and R. Keunings. Optimized partitioning of unstructured computational grids. *Int. J. Num. Meth. Engng.*, 38:433–450, 1995.

[vLA87] P. J. M. van Laarhoven and E. H. L Aarts. *Simulated Annealing: Theory and Applications.* D. Reidel Publishing Company, Dordrecht, 1987.

[vN66] J. von Neumann. A system of 29 states with a general transition rule. In A. Burks, editor, *Theory of Self-Reproducing Automata*, pages 305–317. University of Illinois Press, 1966.

[Wal95] C. Walshaw. A parallelisable algorithm for optimising unstructured mesh partitions. Technical Report P95/IM/03, School of Computing and Mathematical Science, January 1995.

[WCE+95] C. Walshaw, M. Cross, M. G. Everett, S. Johnson, and K. McManus. Partitioning & mapping of unstructured meshes to parallel machine topologies. In A. Ferreira and J. Rolim, editors, *Proc. Irregular '95: Parallel Algorithms for Irregularly Structured Problems*, volume 980 of *LNCS*, pages 121–126. Springer, 1995.

[Wil84] Ray Wild. *Production and Operations Management.* Holt, Rinehart and Winston, 1984. 3rd ed.

[Wil90] R. D. Williams. Performance of a distributed unstructured mesh code for transonic flow. Technical Report C3P 856, California Institute of Technology, January 1990.

[Wil91] Willis. Distributed finite element calculations on transputer arrays and the DAP. *Computing Systems in Engineering*, 2(4):421–424, 1991.

[ZC90] Hans Zima and Barbera Chapman. *Supercompilers for Parallel and Vector Computers.* ACM Press, New York, 1990.

[Zie77] O. C. Zienkiewicz. *The Finite Element Method, 3rd Edition.* McGraw-Hill, London, 1977.